Toward Correctness in the Specification and Handling of Non-Functional Attributes of High-Integrity Real-Time Embedded Systems

Daniela Cancila, Roberto Passerone, *Member, IEEE*, Tullio Vardanega, *Member, IEEE*, and Marco Panunzio

***Abstract*—In high-integrity systems, the focus of the development process is geared to assuring that the assertions made on the system are both correct (i.e., semantically sustainable) and feasible (i.e., true at run time). Some of those assertions take effect in the non-functional domain, that is, in how the system is realized and behaves in time, space and communication during execution; others in the functional domain, and thus concern what outputs the system produces for its inputs. In this paper, we address the problem of achieving correct specification and handling of non-functional attributes, with particular regard to the concurrent structure of the system, the safeness of the interaction protocols engaged in it, and the guarantee that its timing feasibility can be statically verified. Our approach is based on a Model-Driven Engineering methodology, in which correctness can be ensured by construction or verified at a high level of abstraction, while the runtime implementation structure and code are automatically generated. We employ the Ravenscar Computation Model (RCM) and focus, in particular, on aerospace applications, which impose stringent requirements on correctness properties. We discuss an algebraic formalization of our model based on graph theory which we use to prove safe termination in systems compliant with RCM, and show how to use the MAST+ static analyzer to verify the timing aspects. We finally illustrate the results of a prototype tool that was developed for evaluation by major industrial players in the European space industry.**

***Index Terms*—Formal methods, high-integrity real-time embedded systems, model-based engineering, non-functional attributes, ravenscar computational model (RCM), static analysis.**

I. Introduction

OVER the last few years, a wealth of methodologies, techniques and formalisms have been proposed to construe component-based development as a promising approach to address the increasing complexity of large-scale systems [1]–[4]. In most cases, emphasis is placed on those aspects that permit to leverage reuse and separation of concerns, and have to do with quality, integrity, and functionality. In this context, it is a well accepted fact that dependence on *a posteriori* verification is not economically viable. For this reason, considerable effort is being deployed in the industrial and academic communities to establish correct specification and handling of non-functional attributes, with particular regard to the concurrent structure of the system, the safeness of the interaction protocols engaged in it, and the guarantee that its timing feasibility can be statically verified. Of course, a certain amount of verification still has to be performed at the end of the development process. In the embedded systems industry, for example, hardware-software integration and end-to-end functional qualification, which are critical stages of the process, can only occur when all components are ready and the system that integrates them can actually be operated. Even in these cases, however, a rigorous design methodology can help establish a correct implementation path early in the design flow.

In this work, we are interested in studying methodologies for the design of high-integrity real-time embedded systems, and focus in particular on the domain of aerospace applications. Specifically, we target the design of the software architecture for real-time platforms based on Ada and the Ravenscar profile [5]. In this context, we address the problem of augmenting component reuse by separating the treatment of functional and non-functional concerns.

We go about this separation by stipulating that the functional domain is concerned with what outputs the system shall produce for its inputs, whereas the non-functional domain with how the system is realized and behaves in time, space and communication during execution. This taxonomy is no different from that proposed by Glinz [6] and others in a similar vein, except that we restrict functional requirements to solely regard the sequential element of component specification and their static aggregation and interconnection.

The fundamental intent of this separation is to extend the opportunity of software reuse: arguably in fact, the more a functional specification is free from non-functional concerns in the above connotation, the more it can be reused across distinct system models—and thus take on varying non-functional requirements—and/or composed in other functional specifications.

Focus in this paper will be placed in particular on how we handle non-functional properties, such as (explicit and/or implicit) attributes set on the concurrent structure and on the timing behavior of the system, and prove their consistency and assure the correctness of their realization by automatically constructing an implementation from a specification, and by adopting formal

Manuscript received June 03, 2009; revised November 12, 2009; accepted January 29, 2010. First published March 11, 2010; current version published May 05, 2010. Paper no. TII-09-06-0100.

D. Cancila is with CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, Point Courrier 94, Gif-sur-Yvette, 91191, France (e-mail: daniela.cancila@cea.fr).

R. Passerone is with the Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento, 38122 Povo di Trento, Italy (e-mail: roberto.passerone@unitn.it).

T. Vardanega and M. Panunzio are with the Dipartimento di Matematica Pura e Applicata, University of Padova, 35121 Padova, Italy (e-mail: tullio.vardanega@math.unipd.it; panunzio@math.unipd.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2010.2043741

verification techniques to ascertain their satisfaction at run time. Our main results are concerned with warranting the correctness of the concurrent structure of the system in the way of proven compliance with the Ravenscar profile, safe task interaction in the way of freedom from infinite call cycles, and satisfaction of timing attributes. As far as functional requirements are concerned, instead, we expect that the designer has already provided a "correct" functional specification and therefore do not address this aspect of verification. (The reader shall be advised that in this paper we consistently use the term "structure" to mean the concurrent, runtime, organization of the system, which we regard as a key non-functional concern. This connotation is thus unrelated to static architectural considerations, which may also take effect in the functional domain.)

We take a model-driven engineering (MDE) approach to system development [7], [8]. As the name suggests, the MDE methodology focuses on the use of models as the primary means for software construction. In MDE, designers use a number of distinct model spaces which allow them to perform software specification at multiple levels of abstraction. With respect to most component-based design methodologies, MDE introduces a new level of abstraction, called *metamodel*, which is used to specify the form of model elements and the relationships that can exist between them. In our approach, designers specify a system using notions such as classes, interfaces, ports, and components. From there, transformations are employed on the user model to automatically generate a new *implementation* model in which runtime executable components correctly realize in terms of the Ravenscar profile the original components in the specification. Finally, code can be automatically generated from the implementation model.

In an unrestricted general model space, problems may arise from the composition of components, which may cause unexpected, unwanted or erroneous behavior to emerge at run time. Examples are the introduction of invocation cycles which could lead to erroneous (e.g., not terminating) task interactions, or system composition causing tasks to miss their user-defined deadline. These problems can be detected and avoided by fully defining components by their *assumptions* and *guarantees* [9]–[12]. Broadly speaking, guarantees are the services which are provided by a component to its environment; assumptions, instead, are those services which are required by a component from its environment to accomplish its guarantees. In this paper, we address the problem of ensuring that the assumptions which are exposed by a component are satisfied by the guarantees of other components, where satisfaction implies correct synchronization and timing. There are two distinct aspects to this problem: one requires that an assumption (a required service) may connect to a guarantee (a provided service) as long as the guarantee satisfies the assumption *locally*. In this case, the guarantee is sufficient to directly discharge the assumption of the calling component. The other extends this requirement *transitively* across the whole chain of locally-correct assumption-guarantee connections, so that guarantees from components further down in the call sequence can be used to discharge the assumption. The former is a comparatively simple problem of value matching (and possibly round-trip refinement). The latter is considerably more complex because we must infer global properties on the system from guarantees given on individual components. In the context of the MDE methodology, this is further complicated by the fact that the designer specifies the system at an abstract level, using classes and interfaces, while the implementation in terms of threads reside in runtime components, obtained via model transformations.

We adopt the approach proposed by Vardanega [13], and define one metamodel for both the design and the implementation levels. The underlying idea is to use components and interfaces to capture non-functional properties, and to reason, organize and specify the architecture of the system. We then automatically aggregate certain basic elements to build runtime components that comply with the Ravenscar Computational Model (RCM) [5], [14] and satisfy the given system architecture specification. We show that the process enforces certain properties of interest in our domain of application, such as the guarantee of safe termination for all computation threads, and apply static analysis and round-trip engineering techniques to verify the temporal attributes. The overall process has been implemented in a tool based on Eclipse and developed at the University of Padova (Italy). Temporal analysis is performed using MAST+, an extension of the MAST tool originally developed at the University of Cantabria (Spain) [15], also realized at the University of Padova.

II. Related Work

The large increase in the deployment of software systems in recent years have made a range of new devices that integrate several complex functionalities possible. Against this backdrop, the academic and industrial communities have looked at software reuse and component-based techniques as the winning strategy to improve design productivity and to increase quality. Reuse strategies typically exploit a set of pre-designed architectural components and aggregate them by designated composition rules meant to ensure that reuse occurs as intended. In this context, several methods can be used to guarantee correctness of the final result, operating either *by construction* before the realization of an implementation, or *by verification* of the generated system. We take a mixed approach, and deal with structural properties mostly by construction, and with temporal properties by early system verification. Because the literature in this area is vast, we discuss in this paper what we regard as representatives of classes of approaches that are related to our work. Structural correctness is addressed in this section, while work related to temporal attributes is deferred to Section V.

Correctness by construction is described by Chapman as an "economical method to develop security and safety-critical applications" [16]. The fundamental observation is that the later errors are detected, the higher the cost to remove them [17]. Therefore, the underlying idea in a correct by construction flow is to make it hard to introduce errors early in the development and, otherwise, to detect them and remove them as soon as they are introduced using static analysis techniques. Sifakis describes correctness by construction as based on two principles: composability, which ensures stability of component properties across integration, and compositionality, which permits to infer global system properties from component properties [18]. Our approach follows the general idea of correct by construction

design while integrating the process in the MDE development paradigm, endorsed and promoted by the Object Management Group (OMG) [19], [8]. Although our approach is different from that of Sifakis, we arrive at the same conclusion: correctness by construction depends on compositionality and composability. In this work we deal primarily with the correct assembly of the runtime components, hence focusing on our interpretation of compositionality, and we refer the reader to [20] for a discussion on composability.

Component-based techniques can be classified as *abstract* or *constructive* [21]. Abstract approaches are often based on abstract algebra and are typically able to model highly general systems. For example, Montanari *et al.* use graph theory to model the architectural structure of a system [22]. A set of graph transformation rules is used to reconfigure the system and to prove its correctness. Similarly, Reo uses category theory to analyze the coordination of components via mobile channels [23]. Our approach differs in that we restrict our attention to only a class of models, which are suitable for our chosen domain of application. In fact, although our formalization uses graph theory, our results do not generalize to all systems, but apply only to those that comply with RCM [5], [14]. These restrictions allow us to derive specific results, which are not true for the general case.

An intriguing approach to model real-time embedded systems is discussed by Easwaran *et al.* who focus on modeling hierarchical systems out of component interfaces [24], [25]. A component is viewed as an interface decorated by temporal attributes. Hierarchy is used in a bottom-up fashion to understand how time needs to be allocated to the parent interface in order to satisfy the requirements of the children interfaces. In our approach, instead, we treat the system as a flat collection of interfaces and derive results on the *structure* of the overall composition. In particular, we do not use the principle of compositionality to study temporal requirements, which can be verified in our system by static analysis. Hierarchy, if needed, can be modeled using the notion of partition, while temporal budgets can be introduced as discussed in [26]–[28].

Unlike abstract approaches, constructive approaches are based on a specific language or platform which can execute a given set of runtime components. BIP [29] is a typical example of constructive approaches. In BIP, the description of the system is partitioned into three layers. At the bottom layer, the components are described as state machines. Their possible synchronizations are specified in the form of connectors on the second layer. A third layer of priorities restricts the interaction space to obtain deterministic execution. Unlike BIP, which is based on a concurrent state-based model, we use a caller/callee dependency relation in a sequential execution context. While this choice somewhat limits the expressiveness of our model, it also avoids the exponential growth in the number of possible interactions as new connectors are added to the system [29], thereby facilitating analysis.

The work by Lundqvist *et al.* [30] focuses on fault tolerance in high-integrity real-time systems. The approach adopts the Ravenscar Profile to ensure static analyzability of the system in the time dimension. Unlike our approach, timeliness properties are analyzed by "a-posteriori" verification with model checking. The approach further provides runtime hardware monitoring of timing properties (task worst-case execution time and deadline) which does not alter the original timing behavior of the system as in software-based approaches. Timing faults are handled by triggering degraded operational modes for faulty tasks. The execution platform is a Ravenscar-compliant kernel that has been formally verified to ensure the correctness of its behavioral semantics [31].

Several frameworks operate at a higher level of abstraction to assist the designer with the creation and transformation of generic models of the system. One example is the Generic Modeling Environment (GME) [32], which allows the designer to construct and manipulate domain-specific modeling languages. This approach is related to ours in so far as languages can be manipulated to implement a variety of model transformations based on standard traversal patterns or on graph rewriting rules. These are used to automatically convert models between languages, or to generate implementation models. Unlike GME and other tools, which focus on metamodeling technology, our main concern is to prove the correctness of such transformations on our specific choice of model. Other approaches, such as Ptolemy [33], Balboa [3], Metropolis [34] and SystemC-H [35], focus primarily on functional aspects, and provide extensive support for simulation and validation. Instead, we restrict our attention to structural properties and in particular to the interaction protocols between components, generated according to the designer specifications.

To prove our correctness results we rely on a Domain Specific Language (RCM Metamodel) that conforms to the Ravenscar profile. However, the current level of integration in embedded systems is forcing academic and industrial communities to adopt more Domain Specific Languages for a single design. For instance, the MeMVaTEx and Saturn projects [36], [37] use the SysML and MARTE profiles [38], [39] to construct a Domain Specific Language, which separate the modeling of requirements from that of the functionality. Model transformations can be used to integrate the heterogeneous parts, and can be supported by tools such as MDATC [40] to ensure the correctness of the operation. Our choice of a single restricted Domain Specific Language (RCM Metamodel) clearly makes it easier for us to derive specific results, to the detriment, of course, of flexibility in the specification.

Currently, there is a lively debate in the scientific community as to whether to adopt several Domain Specific Languages and then interface them (*heavyweight approach*), or to have only one metamodel (for all the application domains) and to generate several Domain Specific languages from it (*lightweight approach*) [41]. The main advantage of the heavyweight approach is to build a Domain Specific Language that is exactly tailored for the intended aim. However the main drawback is the need to relate the various (meta)models and to integrate the corresponding support tools. The lightweight approach does not incur such a drawback. This approach introduces only one metamodel, which contains general and common notions (such as class or interface) and then builds Domain Specific Languages as a suitable combination of profiles of that metamodel.

We make use of the heavyweight strategy to best address the needs of high-integrity systems. More precisely, we introduce a Domain Specific Language (the RCM Metamodel), which is

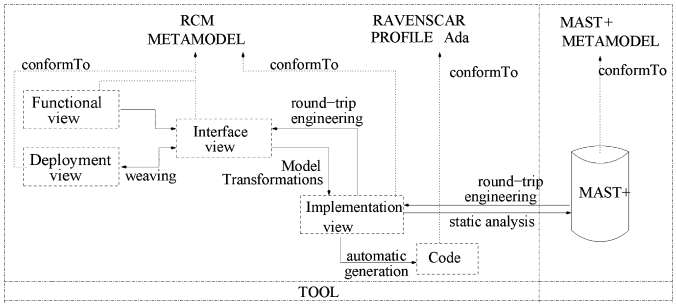


Fig. 1. Overall methodology.

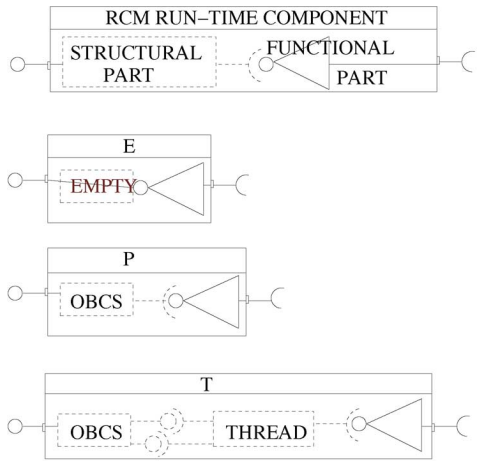


Fig. 2. Structure of runtime RCM components.

tailored to the needs of aerospace applications, and introduces limitations that may not be appropriate in other industrial domains. The extension of our results to a heterogeneous setting is part of our future work.

III. Background and Methodology Overview

We develop our methodology using RCM. RCM is a collection of concurrent components designed to satisfy the restrictions of the Ada Ravenscar profile, which is a subset of the Ada programming language tailored to high-integrity real-time embedded systems [5]. In order to warrant determinism and predictability, RCM prohibits task synchronization, any form of recursion, and dynamic creation or allocation of resources. Moreover, RCM requires tasks to be nonterminating "to mitigate the hazard that may be caused by tasks terminating silently" [5]. Their jobs, on the other hand, do terminate as we will show in Section IV-B. System development in the RCM model space follows a design-by-refinement flow in which an initial abstract functional view is progressively transformed into a detailed architectural description of the system as a composition and interconnection of runtime components. We can place this development process in the broader context of refinement-based methodologies, such as platform-based design [4], [2], by identifying several levels of abstraction (see Fig. 1).

At the more abstract level, represented by the *Functional view*, the designer specifies the functionality of the system as a collection of classes and interfaces using a UML-like notation. Interactions between components are made explicit by identifying functional dependencies between methods. This process partitions the interface services exposed by each component into two groups, those that are *provided* by the component (PI) and those that are *required* from other components (RI) to fulfill the obligations associated with the provided interface. To each PI there corresponds a set of RIs, which model the calls of the PI to other services in the system. Thus, at this level of abstraction, the system is represented both functionally and structurally. However, the interaction between components is limited to a dependency relation and ignores all aspects related to synchronization and timing (which we refer to as attributes on the "system structure" and on time).

The first refinement step consists in decorating PIs and RIs with attributes that qualify the associated method with guarantees (for the PIs) or assumptions (for the RIs) which must be satisfied at run time. We call this level of abstraction the *Interface view* of the system. Refinement actions in this view leave the structure of the system, i.e., the services and their interconnections, and the attributes of the Functional view, unchanged, so they simply add information. The attributes that are introduced are of two kinds. The first has to do with runtime characteristics, such as the worst-case execution time and the period of an operation. The second specifies the desired interaction semantics between interface calls, such as, for example, a periodic clock-based activation or the synchronization protocol. The attributes specified in the Interface view have a direct effect in the next refinement step which consists in automatically generating an *Implementation view*, constructed through a set of model transformations by using appropriate runtime components, which are RCM compliant by definition. In practice, designers work exclusively at the Functional and Interface level, by specifying the desired behavior through a suitable combination of attributes, which are automatically turned into an implementation. The structure of the runtime component and the way the elements are aggregated is determined automatically by the attributes set on the PIs in the Interface view. How this aggregation happens without incurring semantic distortion, i.e., how an RCM component realizes at run time the intentions that the designer specified in the interface attributes, has been formalized in [20]. In actual fact, this is the key to our approach: all that the user is allowed to specify in terms of non-functional attributes can be fully understood by our model transformations, correctly implemented against the constraints of the Ravenscar profile, and correctly verified for timing feasibility by means of static analysis. The reader shall note that the notion of correctness in the latter use above signifies that what is provably asserted of the user model also holds for the implementation model and *vice versa*.

Runtime components in RCM are statically defined and can be of different kinds: *cyclic components*, which are periodically activated by the system clock; *sporadic components*, which are activated by other components or by hardware or software interrupts; *protected components*, which provide data access with exclusion-synchronization guarantees; and *passive components*, which offer data access with no synchronization guarantees. The general structure of a runtime component is shown in Fig. 2. Each component exposes its set of PIs and RIs. The PIs are

in turn decorated with attributes on functionality (visibility and signature), on the structure (e.g., cyclic or sporadic), and on time (e.g., deadline or period) [20], which derive from the attributes selected in the Functional and Interface view. Within an RCM component, every single RI descends from a given PI and the relation between a PI and its RIs is one to many.

Each runtime component has a functional and a structural part. The functional part is a set of *Operational Control Structures* (OPCS) which represent the functional and sequential specification associated to each PI and which specify the RIs needed by the PIs, if any. The structural part regulates the access to the runtime component and it corresponds to an optional protocol and possibly a thread. The protocol (*Object Control Structure*, OBCS in the figure[1]) determines what guarantees (e.g., none, exclusion synchronization, avoidance synchronization) the component offers to the callers of its PI upon access to the static data storage of its functional part. The optional thread, on the other hand, executes a method provided by the component on behalf of the actual caller. The activation of the thread is always mediated by the protocol through a special provided interface reserved for this purpose. If the component includes no thread, then the caller directly executes the service provided by the component, with or without synchronization guarantees depending on the nature of the component. By RCM constraints [5], an RCM component has at most one thread.

While the structure of the Implementation view is generated to guarantee certain properties (which will be discussed in Section IV-B), the temporal properties are statically and automatically analyzed using MAST+ [15]. The results of the analysis are propagated back and made available to the designer in the Interface view. The designer can then change the timing attributes and iteratively reanalyze the system until all temporal requirements are met.

In order to properly generate the final code, the designer must specify the physical structure of the system, i.e., the computational nodes and their interconnections, in the *Deployment view*. At this level, the designer also specifies the logical structure of the system, by aggregating the components of the Interface view in one or more *partitions*, which corresponds to a fault-containment region. He/she then deploys the logical structure on the physical one, by specifying which computational node executes which partition, and the system is scheduled using hierarchical scheduling techniques (see Section V).

Fig. 3 summarizes the attributes of the Functional, Interface, and Implementation view and shows them in relation to the classic V-model.

Table I complements Fig. 3 by summarizing which kind verifications we apply to individual views as well as in the transitions among them.

In the next two sections we describe these attributes in detail as well as the verification techniques we use to ensure correctness.

IV. Structural Properties

We first analyze the structural properties of a design and show that every thread activation always completes without entering infinite cycles of service invocation. To do so, we build a formalization of the model which is more convenient for reasoning. Later, in Section V, we address the problem of checking timing and performance attributes.

[1]The names OBCS and OPCS derive historically from HRT-HOOD [42].

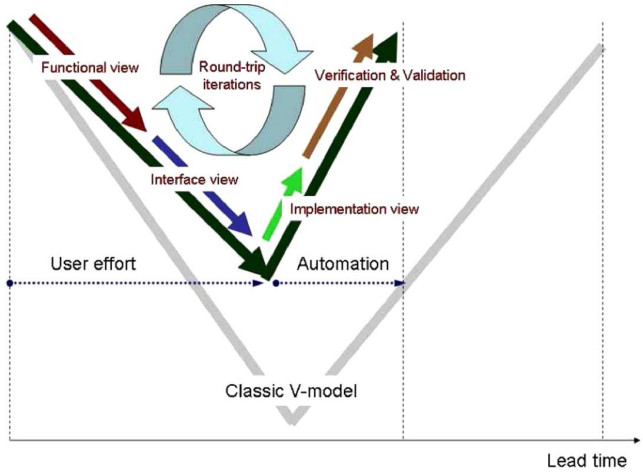


Fig. 3. Design flow with views, attributes and verification techniques.

TABLE I
Relation Between Modeling Views and Verification Steps Taken in Our Approach

VIEW	VERIFICATION
From Functional view to Interface view	Preservation of functional semantics by hypergraph fully-faithful extension
Interface view	Correct task behaviour and interactions by non-functional attributes setting
From Interface view to Implementation view	Full semantic preservation by RCM Interface Grammar
From Implementation view to Interface view	Static feasibility and sensitivity analysis and round-trip

A. RCM Mathematical Foundations

We define a mathematical formulation based on graph theory for the Functional and Interface views. We then use this formulation to prove correctness properties in Section IV-B. We illustrate our formalism through the example shown in Fig. 4, a high level representation of a simplified satellite subsystem which has been proposed, experimented with and successfully tested in the ASSERT project by two major European space industries [43]. In the figure, Position Store (POS) is a data storage, used in read/write mode by Guidance, Navigation and Control (GNC) and Telemetry and Telecommand (TMTC). TMTC reacts to two possible ground commands: it either writes to POS a new value, or it sends a boost correction request to Propulsion (PRO). GNC first reads the current value stored in POS; it then computes the required adjustment and finally updates the initial value accordingly.

1) Functional View: In the Functional view, the designer specifies the functional services provided by system components by using a formalism similar to a UML class diagram. Fig. 5 shows an example of part of the Functional view for the Satellite system, illustrated in Fig. 4. (Here, and in the following, we generally do not show the names of the RIs, to avoid clutter, as these details are usually not important to the discussion.) In

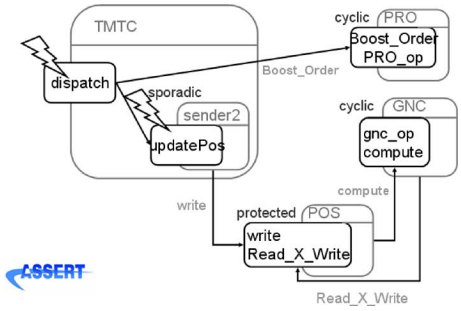


Fig. 4. Satellite system.

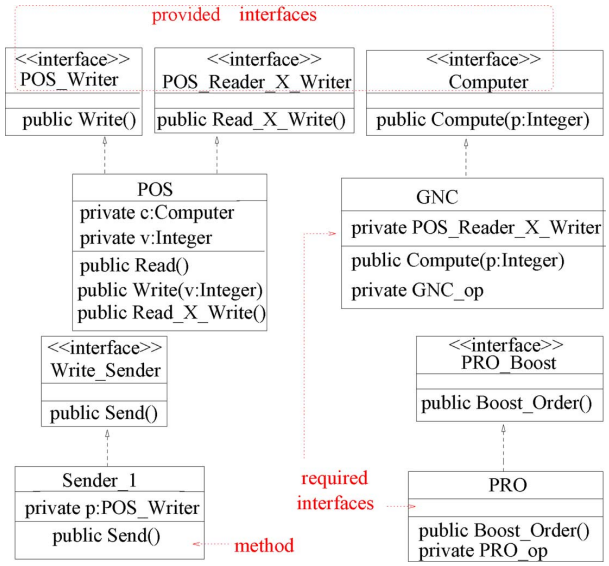


Fig. 5. Functional view for the Satellite system.

the following, we use I_P to denote the set of PIs specified in an RCM system, and I_R for the set of RIs.

We characterize the Functional view as a weighted directed hypergraph $\mathrm{HG}_F = (V_F, E_F)$, where V_F (vertices) is a finite set of services and E_F (hyperedges) are relations over services. Fig. 6 shows an example hypergraph. The set of vertices V_F is obtained as the union of the set I_P of provided and the set I_R of required interfaces in the system, i.e., $V_F = I_P \cup I_R$. Each $PI \in I_P$ and $\mathrm{RI} \in I_R$ has a weight specified by a combination of attributes which collectively characterize the assumptions and guarantees attached to the request and provision of services in an RCM system [20]. The attributes at this level include:

- the *signature* of the method;
- the *state*, or the static variables operated on by the method;
- the *visibility* $\mathcal{V}$ of a method (public, private or restricted);
- the *local WCET (worst case execution time)* of a method, excluding the time required by RIs that may be involved in the execution of the PI.

We write PI attribute. to denote that PI includes weight attribute. The RIs at the level can only be decorated with the signature, which is checked for consistency with that of attached provided service, and with the number of times that the corresponding PI invokes the RI (required to compute the global execution time of the PI).

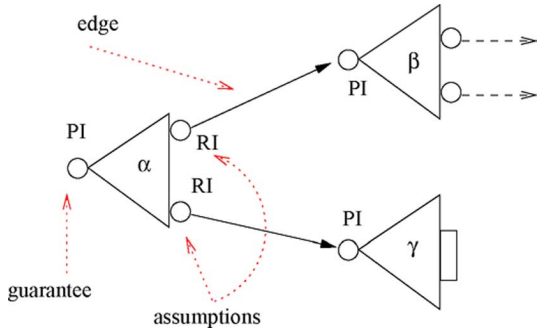


Fig. 6. A hypergraph.

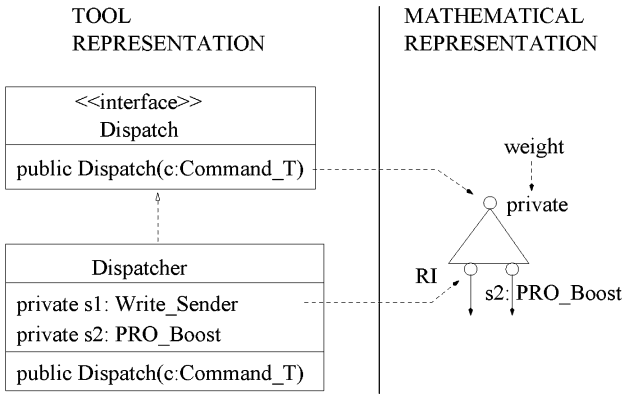


Fig. 7. Functional view and corresponding hypergraph for Dispatcher.

The set of edges E_F is made of two disjoint subsets: hyperedges Δ and simple edges X. A hyperedge $\delta \in \Delta$ relates a single PI to a (possibly empty) list of RIs. We denote a hyperedge δ that links vertex PI with vertices $\mathrm{RI}_1, \ldots, \mathrm{RI}_k$ as

$$\delta = \{\mathrm{PI}, \langle \mathrm{RI}_1, \ldots, \mathrm{RI}_k \rangle\}.$$

The hyperedge identifies the dependency between a provided service and its required services. Hyperedges are depicted as triangles in Fig. 6. For instance, hyperedge α links one PI to a list of two RIs, whereas γ links one PI to an empty list of RI. In Fig. 6, we use a rectangle to represent an empty list of RIs and a circle to represent a PI or a nonempty RI. The list of RIs specifies what services are needed from other components to fulfil the guarantees of the PI.

Conversely, an edge $x \in X$ links a single RI to a single PI and represents the binding of a required interface (function call) to its corresponding provided service. An edge is denoted simply as

$$x = \langle \mathrm{RI}, \mathrm{PI} \rangle.$$

Edges are shown in Fig. 6 as arrows.

Hypergraph HG_F is constructed from the Functional view to formalize the components and their structural relationships as specified by the designer. To illustrate these concepts, consider Fig. 7 which shows the Functional view of the Dispatcher component on the left. Class `Dispatcher` realizes interface `Dispatch` and publishes two private required methods: `s1:Writer_Sender` and `s2:PRO_Boost`.

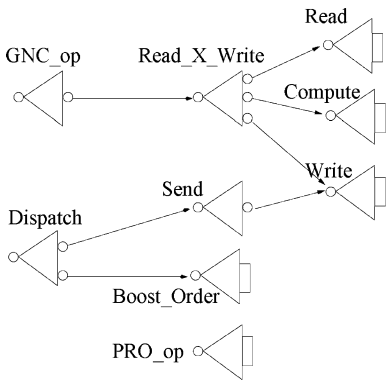


Fig. 8. Hypergraph for the satellite system.

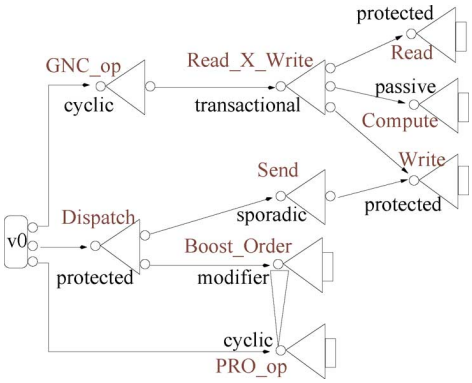


Fig. 9. Hypergraph for the Interface view for the satellite system.

On the right, the figure shows the corresponding mathematical representation. A hyperedge links `Dispatch` to RI `s1:Writer_Sender` and `s2:PRO_Boost`. The other edges link the RIs and PIs to some other interface in the system.

The hypergraph corresponding to the system of Fig. 5 is shown in Fig. 8.

2) Interface View: We represent the Interface view as a hypergraph HG_I which extends HG_F with the addition of the extra set of attributes contained in the Interface view and with an explicit representation of the external environment which is expected to drive the execution of the system. Fig. 9 shows excerpts of the Interface view for the example system. Here, we introduce a new vertex v_0, a new set of hyperedges ψ and additional attributes on the structure and on the time (the temporal attributes, described below, are not shown on the figure to avoid clutter). Formally, $\mathrm{HG}_I = (V_I, E_I)$, where $V_I = V_F \cup \{v_0\}$ and $E_I = E_F \cup X' \cup \Psi$. Vertex v_0 represents the underlying interrupt service infrastructure of the runtime platform, which invokes all $\mathrm{PI} \in I_P$ whose activation depends on external events. For example, all PIs that have a cyclic attribute are activated (enabled for execution) by the clock interrupt and therefore, in our formalization, are activated by v_0. In our case, also the protected PI `Dispatch` is controlled by v_0, since it receives the activation from a ground signal. The additional edges in X' represent these bindings, which are all of the form $\langle v_0, \mathrm{PI} \rangle$ for some $\mathrm{PI} \in I_P$.

We introduce several additional attributes in the Interface view. The designer must select exactly one attribute from the following list:

- *passive*, which identifies a PI that provides no synchronization guarantees on access to data, executing in the caller thread;
- *protected*, which identifies a PI that provides exclusion synchronization guarantees on access to data, executing in the caller thread;
- *transactional*, which identifies a PI that provides a exclusion synchronization guarantees on access to data, executing in the caller thread, transitively across all RIs involved in the execution of the PI;
- *cyclic*, which identifies a PI that is periodically invoked by the clock interrupt;
- *sporadic*, which identifies a PI that is sporadically invoked by a software call;
- *modifier*, which denotes a PI attached to a thread in the same runtime component of another PI, and which takes the cyclic or sporadic nature corresponding to that of the other PI.

In the case of cyclic, sporadic, modifier, the designer further characterizes the PI by:

- the *importance*, which identifies the urgency of the thread;
- the *deadline* of the task;
- the *period of activation* in case of cyclic method;
- the *Minimum Interarrival Time (MIAT)* between two subsequent activations of a sporadic execution.

Besides the attributes set by the designer, other attributes are made available as the result of the analysis aid the user in making design decisions. They are:

- the *ceiling* (priority) of a protected operation; and
- the *global WCET* of a thread, that is, the maximum duration that the execution of the invoked PI may take *including* the cost of any operations invoked by the RIs exposed by the PI;
- the *priority* of a thread.

For the RIs, the attributes include a global WCET which, unlike for PIs, is specified by the designer, and constitutes the assumption made by a component with respect to the maximum running time of the required services. Our tool computes the global WCET guaranteed by the attached PI, and compares the values to verify their consistency. This computation, which involves traversing the graph, is described in Section V.

Let us now elaborate a little more on the rationale for the modifier attribute. In general, the fewer the threads, the easier it is to analyze the system, since the number of possible parallel interactions grows exponentially with the number of threads. A reduced number of threads, however, could make it hard to describe complex systems. To this end, RCM allows the designer to assign the execution of a number of statically defined operations to one and the same underlying runtime component. We model this situation using the set of hyperedges Ψ. Here, we distinguish between the *nominal* behavior of a component, and a finite number of possible alternatives. The PI that corresponds to the nominal behavior, which can be either cyclic or sporadic, is characterized by the attribute nominal. The others are given attribute modifier. Hyperedges in Ψ link the nominal PI to its set of possible modifiers. All modifiers and the nominal operation share the same access protocol (OBCS), whose role is to avoid synchronization between the "calling" thread which invokes a

nominal PI or a modifier and the callee thread which executes that PI.

By definition, hypergraph HG_I is a fully-faithful extension (in the categorical sense) of hypergraph HG_F by an inclusion preserving homomorphism (an embedding), i.e., HG_I preserves the vertices (with their weights), the hyperedges and the edges of hypergraph HG_F. Thus, the only additional information conveyed by HG_I consists essentially of attributes on the structure and on the time of the vertices. Despite this, the distinction between HG_F and its extension HG_I is important. In particular, the same Functional view HG_F may be extended by multiple and distinct Interface views, each specifying alternative communication and synchronization architectures. Different Interface views may also be useful to express different deployments for design space exploration. We use the separation between these two levels of abstraction to help the user maintain a clear distinction between the functionality of the system and the way that it is to be implemented on the execution platform. However, maintaining such separate views is an aspect that we do not address in this work.

To illustrate the Interface view, consider method `Dispatch` in Fig. 7. In the Interface view, this method can be specified for example either as *protected* (to signify that its execution must be performed in a mutually exclusive manner by the caller), or as *sporadic* (to signify that its execution must be performed by a thread of control other than the caller).

3) Implementation View: The attributes set in the Interface views allow us to determine which kind of runtime components realize a given service correctly with respect to both the implementation platform and the specification [20].

The attributes in the Implementation view are all attributes in the Functional and Interface view with the addition of:

- the *Access Protocol* $\mathcal{S}$ to the method (none, mutually-exclusive access, avoidance synchronization);
- the *Concurrent weight* ω, which specifies whether the RCM component, which realizes the method, has a control flow;
- the *Transactional access* τ, which, when set to 1, guarantees transactional access to state(s) across the execution of all of the invoked RIs;
- the *global WCET*, *ceiling* and *priority*, which are also automatically calculated in this view.

a) Model Transformations and RCM Interface Grammar: As described in Section III, there are only three types of RCM runtime components in the Implementation view: passive, protected and threaded. Each runtime component has an external and inner part. Provided and required interfaces characterize the external part. Attributes $\mathcal{S}, \omega, \tau$ specify the inner structural part. For example, in case the designer sets a PI to cyclic, then $\mathcal{S}$ is automatically set to a protocol with guard, ω set to 1 (which means that the runtime component has one THREAD, as per the taxonomy shown in Fig. 2) and τ set to 0 (which means "no transactional access"). The RCM Interface Grammar fully defines model transformations from the Interface to the Implementation view that preserve the intended meaning of PIs in the Interface view [20], [44]. More in particular, we define two formal languages: a Specification Language (L_I) and an Implementation Language (L_C). Both languages share the same set of attributes.

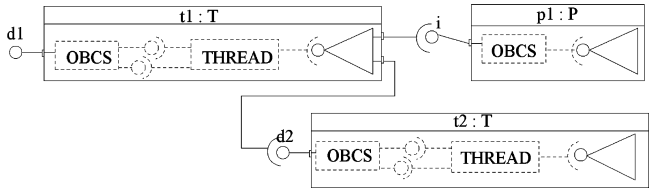


Fig. 10. Function call invocation between RCM runtime components.

Intuitively, a token in languages L_I and L_C represents a type of PI (e.g., cyclic, sporadic, passive, protected). Language L_I only includes terminal token, while L_C includes terminal and nonterminal tokens. We give a set of production and semantics rules in L_C of the form

$$\text{nonterminal token} \rightarrow \text{set of terminal tokens}.$$

The left-hand side of a production rule represents the external part of an RCM runtime component (that is, one type of PI) and the right-hand side represents the inner one. The RCM Interface Grammar is deterministic and non ambiguous, so that given a token in L_I we can always find one (and only one) production rule which identifies the runtime component able to realize the method specified by the designer in the Interface view. Model transformations are fully described by the inclusion of L_I in L_C and the production and semantic rules. The attributes shared between L_I and L_C, and the nonambiguity of the RCM Interface Grammar, ensure that the model transformations incur no semantic distortion [20], hence they are correct with respect to the user specification and the RCM.

b) Message Passing in RCM Runtime Components: The way runtime components interact, and the semantics of function invocation for each class of components in particular, plays an essential role in determining the safe termination of tasks in the system. In order to study the behavior of a thread at runtime level, we distinguish between two kinds of interface services: immediate and deferred.

The set of immediate PIs corresponds to the set of all protected, transactional and passive PIs in the Interface view. Each immediate PI in the Interface view is realized in the Implementation view by an RCM runtime component whose structural part has *no own thread* (component E and P in Fig. 2). Therefore, the thread that invokes an immediate PI executes such PI directly.

The set of deferred PIs corresponds to the set of all cyclic, sporadic and modifier PIs in the Interface view. Each deferred PI in the Interface view is realized in the Implementation view by a *threaded* RCM runtime component whose structural part has both an access protocol (OBCS) and a THREAD (component T in Fig. 2). A threaded RCM runtime components is either a cyclic or a sporadic RCM runtime component, depending on whether the triggering event that releases the thread comes off a hardware clock or a software activity [14].

Consider, for instance, the example shown in Fig. 10, where a component t_1 first invokes $d_2 \cdot$deferred from component t_2, and then $i \cdot$immediate from the protected component p_1. A deferred PI attached to a sporadic thread models an asynchronous data-oriented communication between the caller and the callee.

When the thread of t_1 invokes $d_2 \cdot \mathsf{deferred}$, it deposits the execution request with the OBCS of t_2, which returns immediately, so that the requested service is not executed in the context of the calling thread. Instead, the thread of t_2 is activated as soon as computational resources are available. In fact, following [5], [13], RCM disallows the dynamic creation of threads. Instead, all threads are created at system initialization, and wait for an activation event on a special interface connected to the OBCS [14], [5], for the flow of control of an interface when realized in a runtime components of the Implementation view.

Finally, when the thread of t_1 invokes $i \cdot \mathsf{immediate}$, it *immediately* executes that PI in its own context.

B. Correctness Properties

In this section, we study the structural correctness of the automatically generated system. As discussed in the previous section, attributes on the provided and required services are used to determine the structure of the runtime components [20], [44]. In this section, we analyze the behavior of the threads of the runtime components. We show that, assuming terminating services, the interaction between the components always results in the corresponding jobs to safely terminate. Because services are assumed to terminate, safe termination can be reduced to checking that no cycles are introduced in the call graph. In fact, certain cycles produce no harm, as they involve the activation of different jobs. We show next that we can use the graphs to check these conditions.

To do so, we must analyze all the possible sequences of invocations that are induced by the call graph, which in our case corresponds to the hypergraph HG_I of the Interface view. We start by looking at the possible ways in which each of the PIs in the system might be invoked. We can obtain this information starting from a PI and traversing the edges in the hypergraph backwards to follow the sequence of invocations. We call this a *walk* in the hypergraph.

Definition 1 (Walk): A *walk* is a sequence of the form

$$\mathrm{PI}_0 x_0 \mathrm{RI}_1 \delta_1 \mathrm{PI}_1 x_1 \mathrm{RI}_2 \ldots \mathrm{PI}_k$$

such that $x_i = \langle \mathrm{RI}_{i+1}, \mathrm{PI}_i \rangle$ and $\delta_i = \{\mathrm{PI}_i, \langle \ldots, \mathrm{RI}_i, \ldots \rangle\}$.

In other words, a walk is a path in the hypergraph that represents the transitive closure of calls from service PI_k to service PI_0.

Walks are always finite (by definition), but their length is potentially unbounded, because of cycles, whose presence denounces a nonterminating behavior. This situation in our system is prevented by the design tool, which analyzes the relevant walks in the specification to detect and proactively prevent recursion. We focus on the number of walks to be considered to only those that are potentially harmful. In fact, as discussed in Section IV-A3, when a deferred PI is invoked, the calling thread never executes the sequence of calls originating from the deferred service, which are initiated by the called thread. We can account for this condition by considering only those walks that terminate as soon as they traverse a deferred PI (with the exception of, possibly, PI_0). We call this a *call sequence*.

Definition 2 (Call Sequence): A *call sequence* is a walk $\mathrm{PI}_0 \ldots \mathrm{PI}_k$ such that PI_1 to PI_{k-1} are all $\mathsf{immediate}$, PI_k is $\mathsf{nominal}$, and PI_0 is either $\mathsf{deferred}$, or otherwise an $\mathsf{immediate}$ PI with no related RIs, i.e., there is no $\delta_0 \in \Delta$ such that $\delta_0 = \{\mathrm{PI}_0, \langle \ldots, \mathrm{RI}, \ldots \rangle\}$ for some $\mathrm{RI} \in I_R$.

Thus, a call sequence may include $\mathsf{deferred}$ PIs (which are associated to threads) only at the beginning and at the end of the sequence. In addition, a call sequence is maximal and can not be extended, since if PI_0 is $\mathsf{immediate}$, then it has an empty set of RIs.

In some cases, cycles in a call sequence can be erroneously introduced by designers. However, they are always detected as soon as they occur and reported to the designer that should modify the design until it is cycle-free. Code is always automatically generated from a cycle-free design.

Lemma 1: If a cycle is present in a call sequence, then it is possible to detect it.

Proof: Let $\mathrm{PI}_0 \mathrm{PI}_1 \ldots \mathrm{PI}_k$ be a call sequence. We have two cases.

In the first case, a cycle is present in the call sequence if a PI is traversed twice. This occurs whenever there exist PIs PI_i and PI_j in the sequence, such that $i \neq j$ and $\mathrm{PI}_i = \mathrm{PI}_j$. This condition can be detected by traversing the call sequences originating from all $\mathsf{deferred}$ PIs in a depth-first manner, and verifying if an interface service has already been traversed. This process terminates since RCM has a statically defined number of PIs and RIs. ■

A cycle may still be present if PI_0 is $\mathsf{deferred}$ and $\mathrm{PI}_0 = \mathrm{PI}_k$. This case, however, does not pose any problem since the flow of control is broken by the presence of the OBCS access control structure in the RCM runtime component.

A "harmful" cycle is therefore one that involves only $\mathsf{immediate}$ services. When a cycle is detected, an error is automatically reported in the Interface view and the system is not validated until the cycle is removed by designers.

Given a $\mathsf{deferred}$ PI, we identify a subgraph of the hypergraph that includes all the services reachable from that PI.

Definition 3 (Thread Graph): Let $\mathrm{HG}_I = (V_I, E_I)$ be a hypergraph and let $\mathrm{PI}_T \cdot \mathsf{nominal} \in V_I$ be a nominal PI. A *Thread graph* $\mathrm{HG}_T = (V_T, E_T)$ for $\mathrm{PI}_T \cdot \mathsf{nominal}$ is the subgraph of HG_I such that V_T and E_T include all and only the vertices and the edges that either belong to a call sequence that terminates in $\mathrm{PI}_T \cdot \mathsf{nominal}$ or belong to at least one call sequence that terminates in $\mathrm{PI} \cdot \mathsf{deferred}$ and there is a hyperedge $\psi \in E_I$ such that $\psi = \langle \mathrm{PI}_T \cdot \mathsf{nominal}, \mathrm{PI} \cdot \mathsf{deferred} \rangle$. Like call sequences, harmful cycles in a thread graph are always detected.

Proposition 1: If a cycle is present in a thread graph, then it is possible to detect it.

Proof: The proof follows the same "structure" as that of Lemma 1. Each call sequence in a thread graph is analyzed. Harmful cycles, if any, are then detected by Lemma 1. Note that hyperedge Ψ does not introduce harmful cycles in a thread graph since it only links two deferred methods. ■

Once harmful cycles are detected (if any), errors are automatically reported in the Interface view. The system is not validated until the designer removes all harmful cycles in the thread graph. Therefore, every thread activation always terminates without cycles, and threads are always able to return to their initial condition and be ready to service a new invocation.

We can extend our discussion to a composition of subsystems, in which required interfaces from one subsystem are linked to provided interfaces in the other. Cycles that begin and terminate

at the same deferred PI are again harmless in that they represent the call flow of a sporadic thread whose execution produces the release event for its own next activation. The remaining possible cycles are ruled out by the tool, which immediately invalidates the composition if it detects a cycle in a call sequence within the same thread of control. In this case, the check can be done incrementally, by considering only the call sequences that span across the subsystems. There are, in particular, two cases of interest. In the first, a designer composes two open systems S_1 and S_2 so that each RI of one subsystem invokes a deferred PI of the other subsystem. While cycles may be generated, they are again harmless, because the thread that belongs, for instance, to S_1 never executes directly any of the PIs of S_2, but always proceeds through the OBCS control structure (and vice-versa). In this case, the correctness of the system composition can be inferred compositionally from the correctness of the components, and no further verification is required. The second case is more complex, and involves the invocation of an immediate PI of one subsystem from one RI of the other. The compositionality principle cannot be applied in this case, and the tool is employed to detect and report the presence of potential cycles before the system can be validated.

The attributes set on the Functional and Interface view are sufficient for our tool to conduct the analysis and validate a system. The Implementation view is then automatically generated according to our RCM Interface Grammar [44], which guarantees the implementation of the services with the appropriate runtime components. The generated code is *by construction* guaranteed to be free of harmful cycles (see Proposition 1).

V. Temporal Properties

Structural correctness, as discussed in the previous section, already provides certain guarantees, such as correct structural implementation and task termination. In the following, we will address the problem of ensuring that the *timing* of the tasks satisfies the specification defined by the temporal attributes, by using static analysis. Static analysis, however, critically depends on the assumptions made on execution time and rate of invocation specified in the Interface view. In order to ensure that the stipulations validated in the static analysis stay true at run time we must adopt an execution platform capable of actively monitoring the preservation of runtime properties and thus prevent violations.

A. Execution Platform

The software architecture we target is the *priority-band architecture* [27], a lightweight approach to partitioned systems based on novel features of the Ada 2005 language [45]. Our target platform of choice is GNATforLEON[2] by the Technical University of Madrid, an Ada 2005 cross-compilation system for the LEON2[3] processor.

In the priority-band architecture, the priority range is divided in a set of contiguous nonoverlapping intervals called priority bands. The priority of a task determines the priority band in which it is deployed. The architecture features a two-level hierarchical scheduling. A global fixed-priority preemptive scheduler elects for execution the highest priority band with at least one ready task (active priority band). The choice of the new running task is performed among the ready tasks of the active priority band according to a local scheduling policy which may be fixed-priority preemptive or nonpreemptive, earliest deadline first or round-robin. Each priority band is the implementation of a system partition and the architecture requires to establish a relative order between them in order to stack them in the priority range. The architecture also: (a) enforces the minimum interarrival time (MIAT) of sporadic tasks; (b) monitors with execution-time timers the worst-case execution time (WCET) of tasks to detect overruns; and (c) provides mechanisms to cope with task overruns according to application-specific policies. Hence, temporal isolation between partitions is asserted via schedulability analysis, monitored at run time, and enforced with those mechanisms.

[2]http://polaris.dit.upm.es/~york/

[3]http://www.gaisler.com

B. Timing Attributes

The strand of schedulability analysis theories that are directly related to the Ravenscar Profile are centered around the concept of *periodic workload model* [46] and its extensions (mainly the *sporadic model*). These workload models describe a system as a fixed set of tasks that are executed by the processing resource according to a specific scheduling policy. In the case of the Ravenscar Profile, the adopted form of schedulability analysis is response time analysis [47] for fixed-priority preemptive scheduling (FPPS). In particular, each task is represented by: (i) a period, if it has a cyclic activation pattern, or a MIAT if it has a sporadic activation pattern; (ii) a WCET; (iii) a relative deadline; and (iv) a priority. Shared resources are accessed under the priority ceiling protocol [48], and their minimum *ceiling* can be calculated using the list of resources accessed by each task.

The timing behavior of the system is analyzed using our own specific extension of the MAST tool originally developed by the University of Cantabria, which we named MAST+ [15]. MAST features an event-driven real-time model which has a greater expressive power than the classical sporadic model; for this reason, any system represented in the sporadic model can be represented as a MAST model. To make MAST+ fit our development paradigm and infrastructure we first built a new metamodel for MAST+, and then developed a model transformation that turns the Implementation view into a model that conforms with the MAST+ metamodel. The use of a separate, MAST+ specific, metamodel that is the target and the source of automated transformations from the user model raises the problem of proving semantic preservation across borders. At the current status of our development, correctness is asserted, but not proven. The closeness between the RCM and the event-driven nature of the MAST+ execution semantics makes us confident that our assertion holds. We are however aware that our future use of this infrastructure will require formal proof of this correspondence.

The MAST+ model is used to feed the analysis tool; the results of the analysis are propagated back, first to the MAST+ model, and later attributed to the corresponding RCM runtime components in the Implementation view. Since the transformation that turns the Interface view into the Implementation view is deterministic [20], it is possible to follow its logic backward and propagate the results to the components of the Interface view and show them as attributes of their interfaces (see Fig. 1).

We devised an extension of response time analysis which is based on and extends two independent works on server-based hierarchical scheduling: in [49] the authors devise a schedulability analysis theory for servers with fixed-priority preemptive local scheduling under a FPPS global scheduler; in [50] the authors devise equations for periodic servers with EDF local scheduling under an EDF global scheduler. Our extension [51] is able to account for intra and interpartition shared resources, it is specifically tailored to the priority-band architecture we target [27] and specialized to analyze Ravenscar systems. In fact, the adoption of RCM is very convenient during the timing analysis of the system, since runtime kernels abiding by RCM exhibit a deterministic timing behavior and provide fine-grained timing metrics that accurately characterize the kernel contribution to the timing behavior [14]. We also developed a specialization of holistic analysis that is able to account for those metrics to provide a more accurate form of schedulability analysis for distributed systems [52]. MAST+ was extended to model systems compliant with RCM and equipped with an implementation of those extended schedulability analysis equations.

Let us now discuss the applicable timing attributes and show that we actually *calculate* as many of those attributes as possible during model transformations. The reader shall notice that this automatic calculation is a further contribution to our pursuit of correctness in the handling of non-functional attributes.

WCET information is specified in a descriptor attached to each *functional service* in the Functional view. In order to promote reuse of functional specifications across distinct platforms, it is possible to declare a WCET entry for each distinct execution platform declared in the Deployment view. After the Implementation view is generated, the global WCET of each deferred operation is calculated by summing up the local WCET of operations that belong in the Thread Graph which has that deferred PI as root. This calculation can always be safely performed because call cycles are prevented as discussed in Section IV.B.

A period (or MIAT) and a deadline are specified in the Interface view as attributes of each cyclic or sporadic PI. Those values are later propagated on to the appropriate RCM runtime components upon creation of the Implementation view.

A task priority is an implementation-level attribute and we do not want to pollute the design space with such low-level concerns. We therefore require the designer to only provide a relative order of importance between the deferred operations that belong to components assigned to one and the same partition in the Interface view. When the Implementation view is created, the relative order is used together with the relative order between partitions to calculate the effective priority for the generated RCM runtime component. The Thread graphs are then used to calculate the ceiling priority of the OBCS in cyclic, sporadic and protected components.

The results of the analysis that are propagated to the components in the Interface view comprise the worst-case response time (WCRT) and worst-case blocking time (WCBT) for each periodic or sporadic operation. We decided to show the priority of each periodic and sporadic operation and the ceiling priority of protected operations, even if the designer cannot modify them: albeit those attributes are automatically given in the Implementation view, the designer should be aware of the decisions of the automatic transformations directly in the design space.

TABLE II
WORST-CASE EXECUTION TIME (WCET) OF EACH OPERATION

Operation	WCET	Protected
Receive_TC	0	N
Dispatch	20	Y
Send	15	N
Pro_op	40	N
GNC_op	0	N
Compute	40	N
Read	10	Y
Write	20	Y
Read_X_Write	0	Y

TABLE III
TIMING ATTRIBUTES AND RESULTS OF THE ANALYSIS FOR OUR EXAMPLE. WORST-CASE RESPONSE TIME (WCRT); WORST-CASE BLOCKING TIME (WCBT)

Component	Period / MIAT	WCET	Priority	Deadline	WCRT	WCBT
TC_Receiver	500	20	4	250	20	0
GNC	100	70	3	100	**110**	20
Sender_1	500	35	2	250	195	0
PRO	300	40	1	300	**375**	0

C. Example of Schedulability Analysis

In the following, we perform a schedulability analysis of the example depicted in Fig. 4. The example as described therein represents a closed system; we analyze a scenario which represents the arrival of a telecommand from the ground segment of the system (i.e., its operation center); we thus include an additional sporadic Receive_TC operation, which solely serves to call the Dispatch operation (see Fig. 8). In this specific scenario, the telecommand is then dispatched to the Send operation, so the PRO component performs according to its nominal behavior (there is no call to the modifier operation Boost_Order). First, in Table II, we assign the WCET in milliseconds to all operations. When the entry reads 0, it means that the local cost of the operation is negligible and thus its total WCET is just the sum of the WCET of the operations it calls. For the sake of simplicity, we also assume that the cost to put and fetch requests from an OBCS is null. In a real application scenario however, the modeling of those costs is important, since it may contribute to the blocking factor of tasks.

In Table III, we summarize the threaded runtime components that result from the model transformation of the example, including TC_Receiver, an additional sporadic RCM runtime component originated from the Receive_TC sporadic operation. For each runtime component we list its period or MIAT and overall WCET in milliseconds, Priority, and Deadline.

If we perform a schedulability analysis on this system, we determine that the system is not schedulable. (See Table III.) In particular, the GNC and PRO runtime components would miss their deadlines. For the latter, we may notice that the blocking time of 20 ms, which is part of the worst-case response time, is one of the factors that are making the task not schedulable.

A first possible solution to make the system schedulable would be to reduce the release period of the GNC: a release period of 125 ms would make the system feasible as depicted in Table IV. However, this kind of modification can be quite delicate since it implies a renegotiation of the requirements

TABLE IV
TIMING ATTRIBUTES AND RESULTS OF THE ANALYSIS. WORST-CASE RESPONSE TIME (WCRT); WORST-CASE BLOCKING TIME (WCBT)

Component	Period / MIAT	WCET	Priority	Deadline	WCRT	WCBT
TC_Receiver	500	20	4	250	20	0
GNC	125	70	3	125	110	20
Sender_1	500	35	2	250	125	0
PRO	300	40	1	300	235	0

TABLE V
RESULTS OF THE ANALYSIS WITH THE COMPUTATION TIME OF OPERATION WRITE SHORTENED TO 15 MS. WORST-CASE RESPONSE TIME (WCRT); WORST-CASE BLOCKING TIME (WCBT)

Component	Period / MIAT	WCET	Priority	Deadline	WCRT	WCBT
Ground	500	20	4	250	20	0
GNC	100	65	3	100	100	15
Sender_1	500	30	2	250	180	0
PRO	300	40	1	300	285	0

at system level (in this case it would imply to change the frequency of the GNC control law from 10 Hz to 8 Hz).

In Table V, we propose instead the analysis results for a system in which the WCET of operation Write is optimized from 20 ms to 15 ms, while the rest of the system parameters stay fixed. From the results, we can appreciate that this optimization reduces the overall WCET of both GNC and Sender_1 and since Write is an operation of POS, we also reduce the blocking factor induced by the synchronization protocol on GNC. The combined effect of these factors leads to a schedulable system.

VI. INDUSTRIAL EVALUATION

To assess the viability and effectiveness of the MDE approach outlined in this paper, a team based at the University of Padova (UPD) developed a prototype tool as an Eclipse plug-in. Two industrial teams used and assessed that prototype independently for a total elapsed time in excess of 6 months in three incremental installments. GMF, ATL, and MOFscript [53] were used to develop the engines behind the graphical interface, the model transformations and the code generation, respectively. The full prototype development at UPD took 5.3 person/years from June 2006 to July 2007 to produce: 90 metaclasses, conceptually identical to UML stereotypes, to implement the RCM metamodel common to all modeling views; 13 000 lines of ATL to drive model transformations implemented in accord with the RCM Interface Grammar rules discussed in this paper; 8000 lines of MOFscript to implement code generation; 7500 lines of Java to complete the graphical editor (in addition to 150 000 lines generated automatically by GMF). See [54] for wink clips on how our tools operate. These figures indicate that the magnitude of the effort that was needed to implement our process concept is definitely affordable and that the productivity of the MDE components of the Eclipse environment is convincingly promising.

The prototype tool was used as the baseline development environment by two industrial partners who undertook the redesign of representative subsets of proprietary reference systems. The experimental evaluations conducted by the industrial teams investigated the practicality of the development concept promoted by ASSERT. As all the technology in use was prototypical, attention was not placed on performance factors, but concentrated on the viability of the fundamental principles of the ASSERT vision, in particular:

i) active enforcement of separation of concerns: the designer must be able to concentrate on functional aspects and to (only) declaratively specify the non-functional requirements that must be met by the contractual interfaces of system components;
ii) expressiveness and coverage of the non-functional requirements settable on the contractual interfaces, in order to decrease development time;
iii) proof of correctness of the automated transformations that turn the user model (the Functional and Interface view) into the Implementation view and then into the source code to be submitted to compilation and binding to the platform-specific middleware: the user must be able to place justified confidence in the correctness of the transformation process and in the ultimate economy of the residual stage of verification and validation required on the end product;
iv) ability to explore the solution space in the Implementation view in a round-trip feedback-based manner originating from the designer model space. The interested reader is referred to [52] for details on the forms of static real-time analysis supported in the proposed approach.

Both experiments confirmed that our vision does indeed address the industrial requirements enumerated above and that it actually meets some of them to full satisfaction. In particular: (i) Separation of concerns in the designer space was deemed to be both desirable and achievable. (ii) The declarative specification permitted in the designer space has potential for decreasing the development time considerably by sparing the burden of decomposing the system down to primitive runtime entities and then having to prove their local and global correctness. The reader should notice that this specific observation largely matches the focus of this paper. (iii) The provision of proof support for transformations and verification, in the form, for instance, of the RCM Interface Grammar [20], [44] was considered extremely important, though major effort is required of industrial practitioners to acquire full control of it. (iv) Increase reliance on automation is considered a key asset of the future development style.

The industrial evaluations decreed the success and the strategic relevance of the project vision, including all aspects discussed in this paper, but also concluded that its actual deployment in the current industrial process critically depends on the availability of robust, reliable, and mature development technology. The positive conclusion of the project created sufficient momentum to launch two distinct initiatives aimed to further the project proceedings in complementary directions: (i) the European Space Agency undertook a series of focused investments to strengthen the quality of the ASSERT technology base and make it known to its own set of industrial suppliers: the overarching goal of that initiative is to turn the resulting infrastructure and process into the standard practice for all developments resulting from future ESA procurements,

cf. [43] for more details and (ii) a core subset of the ASSERT project team, among which two of the authors of this paper, instigated a successor project (nicknamed CHESS and funded in the ARTEMIS JU program framework) to pursue two additional goals in a broadened cross section of high-integrity industry, which spans railway, space, telecommunications, and automotive: the extension of the ASSERT approach to address dependability properties; and the preservation of compositionality in the presence of heterogeneous development chains.

In fact, the latter objective forms one of main strands of research that follow from the results discussed in this paper: whereas the approach we presented is essentially built bottom up by being rooted in RCM, so that the user specification can always be understood, realized and validated in terms of semantically consistent runtime components, the intent to make that approach apply across distinct industrial domains require that one and the same user specification should allow for multiple transformations, thus strengthening its compositionality.

VII. Conclusion

In this paper, we have presented a model-driven approach to component-based development whose aim is to enable the correct expression, interpretation, realization, and verification of non-functional attributes in high-integrity real-time embedded systems. We have discussed correctness in terms of safe task interaction protocols, and of fully predictable, hence statically verifiable, execution behavior. We have presented an algebraic formalization of the RCM model space and used it to derive proofs of structural correctness and safe termination. The vision, concepts, and notions discussed in this paper were actually realized in the implementation of a prototype tool that was successfully test-cased in the context of the ASSERT project [43] under the coordination of the European Space Agency and the participation of major space industries, along with small and medium enterprises, tool vendors, research centers and universities across Europe. The reader is referred to [54] for details, including wink clips, on how the prototype tool operates.

One limitation of our work is that transformation rules between the Implementation view and MAST+ are not formalized. The lack of formalization prevents us from proving the semantic integrity of the transformations like we have done between the Interface and Implementation views. We will address this limitation as part of future work.

As we discussed at the end of the previous section, our current work addresses the current industrial trend that demands the integration of heterogeneous functionality within the same system. This requires that our notion of correctness, the concept of compositionality, and the separation between attributes on functionality and on the structure be extended to cross the boundaries between heterogeneous models. For this reason, we are investigating the interpretation of different profiles in the context of the model-driven engineering methodology to model a given system.

References

[1] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 366–390, Mar. 1997.

[2] A. Pinto, A. Bonivento, A. L. Sangiovanni-Vincentelli, R. Passerone, and M. Sgroi, "System level design paradigms: Platform-based design and communication synthesis," *ACM Trans. Design Autom. Electron. Syst.*, vol. 11, no. 3, pp. 537–563, Jul. 2006.

[3] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta, "BALBOA: A component-based design environment for system models," *IEEE Trans. Comput.-Aided Design of Integr. Circuits and Syst.*, vol. 22, no. 12, pp. 1597–1612, Dec. 2003.

[4] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Design of Integr. Circuits and Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[5] A. Burns, B. Dobbing, and T. Vardanega, Guide to the use of the Ada Ravenscar Profile in high integrity systems, University of York, York, U.K., Tech. Rep. YCS-2003-348, 2003. [Online]. Available: http://www.cs.york.ac.uk/ftpdr/reports/YCS-2003-348.pdf

[6] M. Glinz, "On non-functional requirements," in *Proc. 15th IEEE Int. Conf. Requirements Eng.*, Oct. 21–26, 2007.

[7] B. Selic, "From model-driven development to model-driven engineering," Keynote Talk at ECRTS'07. [Online]. Available: http://feanor.sssup.it/ecrts07/keynotes/k1-selic.pdf

[8] D. Schmidt, "Model-driven engineering," *IEEE Computer*, pp. 25–31, Feb. 2006.

[9] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proc. 9th Annu. Symp. Foundations of Softw. Eng.*, 2001, pp. 109–120, ACM Press.

[10] A. Benveniste, B. Caillaud, and R. Passerone, "A generic model of contracts for embedded systems," Institut National de Recherche en Informatique et en Automatique, Rapport de Recherche 6214, Jun. 2007.

[11] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis, "A contract-based formalism for the specification of heterogeneous systems," in *Proc. Forum on Specification & Design Languages (FDL'08)*, Stuttgart, Germany, Sep. 23–25, 2008.

[12] D. Cancila and R. Passerone, "Functional and structural properties in the model-driven engineering approach," in *Proc. 13th IEEE Int. Conf. Emerging Technol. Factory Autom. (ETFA08)*, Hamburg, Germany, Sep. 15–8, 2008.

[13] T. Vardanega, "A property-preserving reuse-geared approach to model-driven development," in *Proc. 12th IEEE Int. Conf. Embedded and Real-Time Comput. Syst. Appl.*, Aug. 2006, pp. 223–230.

[14] T. Vardanega, J. Zamorano, and J. de la Puente, "On the dynamic semantics and the timing behaviour of Ravenscar kernels," in *In Real-Time Systems*. New York: Springer, 2005, vol. 29, pp. 58–89.

[15] M. Gonzáles Harbour, J. J. Gutiérrez, J. C. Palencia, and J. M. Drake, "MAST: Modeling and analysis suite for real-time applications," in *Proc. Euromicro Conf. Real-Time Syst.*, Delft, The Netherlands, Jun. 2001. [Online]. Available: http://mast.unican.es

[16] R. Chapman, "Correctness by construction: A manifesto for high integrity software," in *Proc. 10th Australian Workshop on Safety Critical Syst. Softw.*, , 2006, Australian Computer Society, Inc..

[17] A. Hall, "Realizing the benefits of formal methods," *J. Universal Comput. Sci.*, vol. 13, no. 5, pp. 669–678, 2007.

[18] J. Sifakis, "Embedded systems—Challenges and work directions," in *Principles of Distributed Systems* LNCS, 2005, vol. 3544.

[19] OMG. [Online]. Available: http://www.omg.org/

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

[21] A. Sangiovanni-Vincentelli and R. Passerone, "Contract based formalisms for heterogeneous and hybrid systems." 2007. [Online]. Available: http://www.artist-embedded.org/docs/Events/2007/Components/Slides/RobertoPasserone.pdf

[22] R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto, "Style based reconfigurations of software architectures," Università di Pisa,, Pisa, Italy, Tech. Rep. TR-07-17, 2007.

[23] F. Arbab, "Reo: A channel-based coordination model for component composition," *Mathematical. Structures in Comp. Sci.*, vol. 14, no. 3, pp. 329–366, 2004.

[24] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee, "Incremental schedulability analysis of hierarchical real-time components," in *Proc. EMSOFT 2006*, 2006, pp. 272–281.

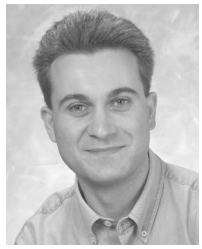
[25] A. Easwaran, M. Anand, I. Lee, and O. Sokolsky, "On the complexity of generating optimal interfaces for hierarchical systems," in *Proc. 1th Int. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2008.

[26] M. Panunzio and T. Vardanega, "A metamodel-driven process featuring advanced model-based timing analysis," in *Ada-Europe 2007*. Berlin, Germany: Springer-Verlag, 2007.

[27] J. A. Pulido, S. Urueña, J. Zamorano, T. Vardanega, and J. A. de la Puente, "Hierarchical scheduling with Ada 2005," in *Proc. 11th Ada Europe Int. Conf.* , Porto, Portugal, Jun. 5–9, 2006, LNCS 4006, pp. 1–12, ISBN: 978-3-540-34663-0.
[28] E. Bini and G. Lipari, "A methodology for designing hierarchical scheduling systems," *J. Embedded Comput.*, vol. 1, no. 2, pp. 257–269, Apr. 2005, ISSN:1740-4460.
[29] S. Bliudze and J. Sifakis, "The algebra of connectors—Structuring interaction in BIP," in *Proc. Int. Conf. EMSOFT*, 2007, pp. 11–20.
[30] K. Lundqvist, J. Srinivasan, and S. Gorelov, "Non-intrusive system level fault-tolerance," *Reliable Software Technologies—Ada-Europe*, pp. 156–166, 2005.
[31] K. Lundqvist and L. Asplund, "A Ravenscar-compliant run-time kernel for safety-critical systems," *Real-Time Systems*, vol. 24, no. 1, pp. 29–54, 2003.
[32] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. IEEE*, vol. 91, no. 1, Jan. 2003.
[33] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Z. , Heterogeneous concurrent modeling and design in Java Univ. California, Berkeley, Tech. Rep. UCB/ERL M05/21, Jul. 2005, vol. 1, Introduction to Ptolemy II.
[34] F. Balarin *et al.*, "Metropolis: An integrated electronic system design environment," *Comput. Mag.*, pp. 45–52, Apr. 2003.
[35] H. D. Patel, S. K. Shukla, and R. A. Bergamaschi, "Heterogeneous behavioral hierarchy extensions for system C," *IEEE Trans. Comput.-Aided Design of Integr. Circuits and Syst.*, vol. 26, no. 4, pp. 765–780, 2007.
[36] MeMVATEX French Project, "Méthodologie Pour la Modélisation, Lavalidation et la Tracabilité Des Exigences (MeMVaTEx)." [Online]. Available: http://www.memvatex.org
[37] SATURN Project, "SysML bAsed Modeling, Architecture ExploRation, Simulation and SyNthesis for Complex Embedded Systems." [Online]. Available: http://www.saturnsysml.eu
[38] OMG, "Systems Modeling Language SysML." [Online]. Available: www.sysml.org
[39] OMG, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Beta 2." [Online]. Available: www.omg-marte.org
[40] R. Bendraou, P. Desfray, M. Gervais, and A. Muller, "MDA tool components: A proposal for packaging know-how in model driven development," *Software and Systems Modeling*, vol. 7, no. 3, pp. 329–343, Jul. 2008.
[41] R. Passerone, I. B. Hafaiedh, S. Graf, A. Benveniste, D. Cancila, A. Cuccuru, S. Gérard, F. Terrier, W. Damm, A. Ferrari, L. Mangeruca, B. Josko, T. Peikenkamp, and A. Sangiovanni-Vincentelli, "Metamodels in europe: Languages, tools, and applications," *IEEE Design & Test of Computers*, vol. 26, no. 3, pp. 38–53, May/Jun. 2009.
[42] A. Burns and A. Wellings, *HRT-HOOD A Structural Design Method for Hard Real-Time Ada Systems*. York, U.K.: Elsevier, 1995, University of York .
[43] ASSERT Project, [Online]. Available: http://www.assert-project.net
[44] D. Cancila and T. Vardanega, RCM Interface Grammar University of Padova, Padova, Italy, Tech. Rep., 2009. [Online]. Available: http://www. math.unipd.it/tullio/ASSERT/RcmInterfaceGrammar.pdf
[45] ISO SC22/WG9, "Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) With Technical Corrigendum 1 and Amendment 1," 2005, .
[46] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[47] M. Joseph and P. K. Pandya, "Finding response times in a real-time system," *Comput. J.*, vol. 29, no. 5, pp. 390–395, 1986.
[48] J. B. Goodenough and L. Sha, "The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks," in *Proc. 2nd Int. Workshop on Real-Time Ada Issues*, 1988, pp. 20–31.
[49] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proc. 26th IEEE Real-Time Syst. Symp.*, 2005, pp. 389–398.
[50] J. L. Lorente and J. C. Palencia, "An EDF hierarchical scheduling model for bandwidth servers," in *Proc. 12th Int. Conf. Embedded and Real-Time Comput. Syst. Appl.*, 2006, pp. 261–266.
[51] M. Panunzio and T. Vardanega, "An approach to the timing analysis of hierarchical systems," in *Proc. 13th Int. Conf. Embedded and Real-Time Comput. Syst. Appl.*, 2007.
[52] M. Bordin, M. Panunzio, and T. Vardanega, "Fitting schedulability analysis theory into model-driven engineering," in *Proc. IEEE Euromicro Conf. Real-Time Syst. (ECRTS 08)*, Jul. 2008, pp. 135–144.
[53] [Online]. Available: http://www.eclipse.org/modeling
[54] D. Cancila, M. Trevisan, and T. Vardanega, "A Gentle Introduction to the HRT-UML/RCM Methodology." [Online]. Available: http://www.math.unipd.it/tullio/~Research/ASSERT/Tutorial

Daniela Cancila received the Laurea master degree in philosophy from "La Sapienza," the first University of Roma, Rome, Italy, and the Ph.D. degree in theoretical computer science from the University of Udine, Udine, Italy, where she has studied with C. Bohm and F. Honsell.

Since 2002, she has been teaching computer science at Italian and French universities. Since 2008, she is a Research Engineer at the "Commissariat à l'Energie Atomique et aux Energies Alternatives" (CEA), France. Her research interests include methodologies, design and tools for model-based safety engineering of real-time systems.



Roberto Passerone (S'96–M'05) received the Laurea degree (*summa cum laude*) in electrical engineering from the Politecnico di Torino, Torino, Italy, in 1994, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1997 and 2004, respectively.

From 1998 to 2005, he was with Cadence Design Systems, Berkeley, CA, where he held various positions from Senior Member of Technical Staff in the System Level Design Product Group, to Research Scientist in the Cadence Berkeley Laboratories. Since 2006, he has been an Assistant Professor with the Department of Information and Communication Technology at the University of Trento, Trento, Italy. His research interests include the design and implementation of high-performance microprocessors, system level design, communication design and formal methods. In particular, his research has focused on the development of methods for the automatic synthesis of protocol converters and transactors, and for the analysis of the semantic foundations of heterogeneous systems.

Tullio Vardanega (M'95) graduated with a degree in computer science at the University of Pisa, Pisa, Italy, in 1986 and the Ph.D. degree in computer science from the Technical University of Delft, Delft, The Netherlands, in 1998, while working at the European Space Research and Technology Centre (ESTEC), Noordwijk, The Netherlands.

At ESTEC, over the period 1991–2001, he held responsibilities for research and technology transfer projects as a lead person in the area of onboard embedded real-time software. In January 2002, he was appointed Lecturer in Computer Science, Faculty of Science, University of Padova, Italy, before becoming Associate Professor in October 2004. At Padova, he took on teaching and research responsibilities in the areas of high-integrity real-time systems, quality-of-service under real-time constraints and software engineering methods, including model-driven engineering, and processes for such environments. He has authored numerous papers and technical reports on these subjects. He runs a range of research projects in these areas on funding from international and national organizations.

Marco Panunzio received the Laurea Specialistica (M.Sc.) degree in Computer Science (*cum laude*) from the University of Padova, Padova, Italy, in 2006. He is currently working towards the Ph.D. degree in computer science at the University of Padova, Padova, Italy.

He is collaborating with the European Space Agency in the scope of the Networking/Partnering Initiative (NPI). His main research interests are schedulability analysis of real-time systems, model-driven engineering, component-based software engineering, and software reference architectures.