	Reference: *UPD-E02-D503-1* **Date**: *18/08/2008* **Issue**: *0.2* **Page**: *1 of 23*

COrDeT - Cannes Study
Functional Modeling with the RCM Methodology

Output of WP503

Written by:	**Organization**	**Approval Status**
Marco Panunzio	UPD	04/08/2008
Tullio Vardanega	UPD	13/08/2008
Verified by:		
Tullio Vardanega	UPD	18/08/2008
Approved by:		
Gerald Garcia	TAS-F/Cannes	pending
Andreas Jung	ESA/ESTEC	pending

Document Change Record

Issue/Revision	Date	Change Record	Author
0.0	04/08/2008	Initial version	Marco Panunzio (UPD)
0.1	13/08/2008	First internal release	Tullio Vardanega (UPD)
0.2	18/08/2008	First external release	Tullio Vardanega (UPD)

Abstract

This report complements the documentation on the HRT-UML/RCM methodology produced in the ASSERT project, by discussing the constraints that hold the modeling space named the "Functional View". The interested reader may find the necessary complementary information in the following ASSERT reports that are published for convenience in the COrDeT repository at URL `http://www.pnp-software.com/cordet/process/methodology.html`:

D3.1.4-1: Guide for Using AP-level Modelling Containers (Part 1), also available as an online tutorial at URL `http://www.math.unipd.it/~tullio/Research/ASSERT/Tutorial/`

This document is a tutorial-level guide to the whole of the HRT-UML/RCM methodology and should possibly be read first in complement to COrDeT Report WP502 (this document).

D3.2.6-1: Refinement of Model Transformation Rules from Use Experience in V2 Demonstrator.

This document illustrates the model-driven transformation-based essence of the HRT-UML/RCM methodology and should be read after D3.1.4-1.

D4.2.1-3: Software Design Tool Prototype (Final Version)

This document presents the first public release of the prototype tool infrastructure that supports the HRT-UML/RCM methodology. This document should be read by those who want to have a more detailed understanding of the engineering activities required to apply the methodology.

D7.3-2: Consolidation of requirements and MW definition and coverage.

This document discusses the constructive principles of the ASSERT Virtual Machine (VM) and analyzes its provisions for isolation against time, space and communication faults. The HRT-UML/RCM methodology requires that an ASSERT VM operates on the target platform to assure that the level of property-preservation guaranteed by the automated model transformations that realize the system implementation extend to run-time execution.

This document should be read in complement to COrDeT Report WP603.

The Functional View is the specific modeling view in which the sequential behavior of the system is specified. HRT-UML/RCM strives to make the Functional View *void* of any non-functional aspects, in particular related (but not limited) to *concurrency* and *time*. Non-functional aspects are addressed in two distinct views, called Interface View and Deployment View, which are presented in ASSERT-specific documents, manuals and tutorials and thus not repeated here. In this report we illustrate the advantages of this choice. We especially emphasize how this approach can facilitate the reuse of functional specifications across multiple non-functional architectures. We also explain in language-neutral terms what constructs should not be included in a language to model a functional specification compliant with the HRT-UML/RCM.

Acknowledgments. The authors of this document gratefully acknowledge the contribution of all members of the UPD unit of the ASSERT project team at UPD, as well as Stefano Puri and Silvia Mazzini of Intecs who also collaborated to the definition and implementation of the Functional View of the RCM methodology.

Contents

Chapter 1

Overview of HRT-UML/RCM

The Model-Driven Engineering (MDE) paradigm strives to raise the abstraction level of the development and to exploit automatic model transformation and code generation for a large part of the production process. However, the application of the MDE paradigm to the development of high-integrity real-time systems requires the adoption of a methodology that can ensure that the properties of interest are expressed at model level and then verified and preserved throughout the entire development process as well as at all levels of abstraction, from the model space(s) via source code to execution at run time.

HRT-UML/RCM [BV07, PV07] is an MDE methodology, with an associated prototype infrastructure, devised in the scope of the ASSERT project (`www.assert-project.net`).

HRT-UML/RCM aims to: (i) provide a design environment in which the user almost exclusively manipulates *platform-independent models* (PIM), with the sole exception of the specification of hardware configuration and the application deployment to it; (ii) realize an instantiation of the general MDE paradigm endowed with principles of *correctness by construction* [Cha06] and property preservation [Var06] from model down to execution; (iii) guarantee that all models produced as part of the transformation process are trustworthy representations of the system as it was intended by the user and as it is to be executed at run time.

In HRT-UML/RCM, the user design space is the PIM, which is free from any concerns related to programming languages, implementation technologies and execution platforms. The PIM must conform to a specific metamodel, which restrains the design space and prescribes syntax, ontology and semantics of all legal PIM entities, their relationships and the constraints that hold on them.

HRT-UML/RCM adopts a *single computational model*, the *Ravenscar Computational Model* [BDV03] (RCM for short). Adopting the RCM we earn several advantages, since systems that comply with it are by definition: (i) amenable to static analysis for time, space and other dimensions of behavior, since they abide by a deterministic model of concurrent execution; and (ii) can also be targeted to small and efficient run-time kernels which may be economically certified for compliance with the intended execution semantics.

As a further point of interest, in contract to other mainstream modeling languages (in particular UML [OMG07b] and the MARTE profile [OMG07a]) the HRT-UML/RCM model space does not allow any semantic variation points: the run-time semantics expressed in its models thus always is completely defined.

An automated model-to-model transformation turns the PIM into a lower-level (i.e., less abstract) *platform-specific model* (PSM), which is compliant with the RCM by construction. In HRT-UML/RCM, the user specification of the PIM is essentially declarative, while the transformation process applied to it corresponds to an implementation designed to be provably correct by construction. The crucial benefit of the latter is of course that an implementation of that kind does not need to be verified a posteriori on a per-system basis, but it only requires a single per-platform validation, with considerable cost saving for the developer.

In contrast with other relevant approaches, like for example those promoted by AADL [SAE] or the MARTE profile [OMG07a], the user is not required to ensure the consistency of all the models developed to represent system views of interest. The modeling space under the control of the user is significantly reduced, which arguably allows more focused verification and thus, presumably, better return on investment.

While multiple PSM may in principle be produced from a single PIM, HRT-UML/RCM has, for now, elected to produce single PSM, which may also be used as a *Schedulability Analysis Model* (SAM). Other PSM of interest may for instance address dependability, safety and security concerns.

The SAM is a formal model that represents the semantics of the system in so far as it allows to statically analyze its timing behavior. The SAM generated in HRT-UML/RCM is comprised of a set of comparatively simple Ravenscar-compliant building blocks (as the RCM is in effect a kind of *Component Model*, those building blocks are in fact "components") which, through correct-by-construction composition, may arrive at encompassing arbitrarily complex execution semantics that realize the user specification as set in the PIM.

The HRT-UML/RCM infrastructure seamlessly integrates round-trip support for timing analysis. In the intent of making the PIM the center of the user modeling space, round-trip starts and ends at the PIM [BPV08]. While the analysis is of course made on the SAM, the results from it are reported back to the PIM, which is possible as the entire model transformation logic is deterministic and reversible, hence it may also be followed backwards.

The HRT-UML/RCM infrastructure also features automated generation of source code. This leg of the transformation process starts from the SAM, which eases the provision of constructive proofs that confirm that the system at run time corresponds to what was analyzed and deemed feasible in the SAM. In our specific context, the complexity of the code generator is modest since the SAM is very close to the system at run time and the code generation engine makes extensive use of simple Ravenscar patterns [PdlPH+07].

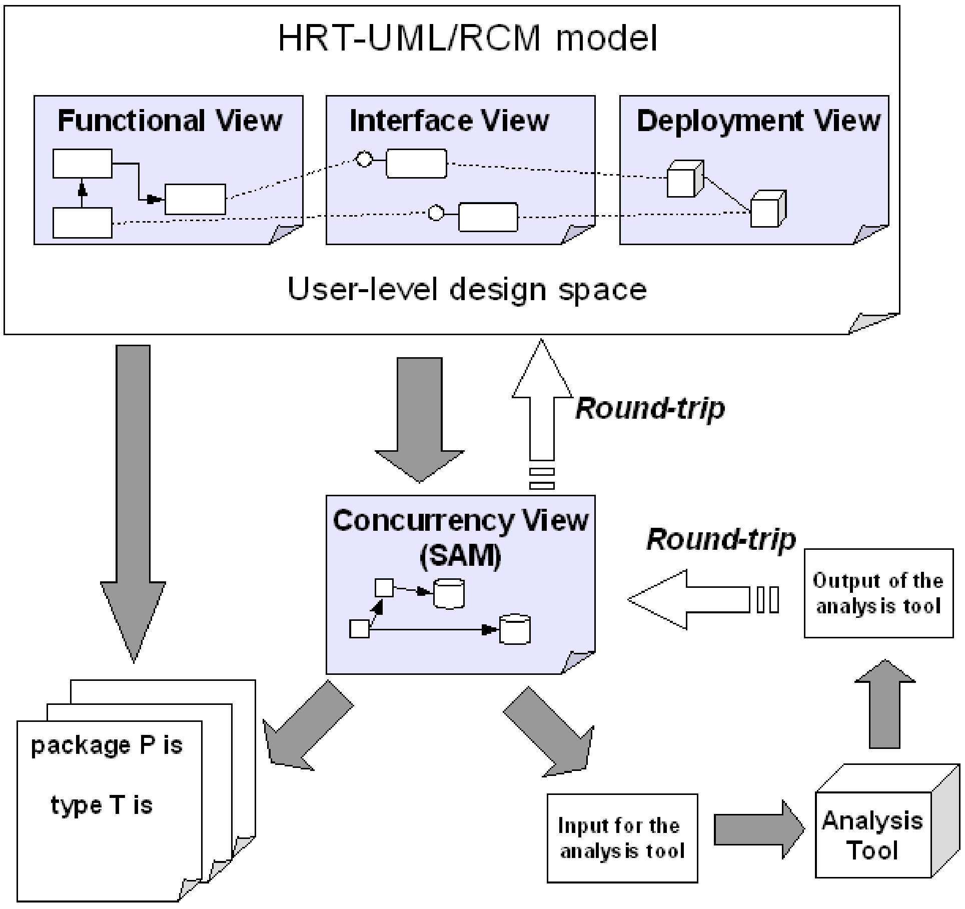

Figure 1.1: Overall process

Chapter 2

A Multiple View Design Environment

It may be argued that the design of high-integrity real-time systems does benefit from some educated form of *separation of concerns*, in particular between the functional and non-functional modeling. Each design view should possess a dedicated design space in which the user may comfortably express the required semantics and perform specialized stages of model-based analysis for verification and fine-tuning. Under this setting, it is of course paramount that adequate mechanisms are in place to ensure that a consistent overall system may result from the composition of each concern-specific view. This is the difficult part of the process. HRT-UML/RCM solves that problem by adopting a single metamodel to underpin all model views, with dedicated "projections" to each of them, so that system components are by definition capable of accepting the attributes set by the user in the applicable views, as well as proven model transformation rules that ensure that component composition always results in a statically-analyzable system.

The reasons why we want to achieve a *controlled* form of separation of concerns are manifold.

Firstly, the design of high-integrity real-time systems requires attention to very different aspects, including but non limited to concurrency, functional specification, hardware selection and modeling, verification and validation, system deployment and configuration: separation of concerns helps the user concentrate only on the aspects of interest at the time. Furthermore it is most definitely possible that distinct designers with different project roles are working on the same system whereby separation of concerns helps each single designer gain full intellectual control over the aspects of pertinence without the need to also have to master the other system views.

Secondly, *control* over separation of concerns, is achieved using a *single* MDE infrastructure enforcing a precise methodology and it is essential to: (i) guarantee the *consistency* between every design concern; (ii) enforce and demonstrate traceability between model entities across views of potentially distinct abstraction levels.

Thirdly, if the design environment provides distinct views of the system, it is possible to adopt in each view the most suitable design notation and it is thus easier to verify each view in isolation, using the most appropriate verification tools.

And finally, the separation between the concurrent architecture and the functional (sequential) behavior allocated on it facilitates reuse of the latter across distinct non-functional architectures as well as, interestingly, the reverse, that is, the reuse of a single concurrent architecture with different functional behaviors (see fig. 2.2).

In HRT-UML/RCM the design environment rests on a *single* underlying metamodel, which integrates and enforces the semantics of each design aspect of interest and provides it in separated but yet related system views. It is clearly of paramount importance to guarantee the consistency between distinct views: modifications performed in a view should be consistently propagated to the other system dimensions upon view switching. We contend that it is too demanding and error-prone to leave the designer in charge of assuring such consistency: rules embedded in the MDE infrastructure enforce *automatically* the sought consistency at each view switch.

In HRT-UML/RCM the design space offered to the user features three views:

- a *Functional View* in which the designer specifies the sequential behavior of the system components by means of class diagrams, state machines, sequence diagrams;
- an *Interface View*, which allows the designer to specify how functional components aggregate together and what services they expose and require to one another. This modeling view is expressed using declarative specifications that decorate the interface of model components and the relations among them with attributes which specify the desired concurrent semantics within the expressive power availed by the underlying RCM metamodel;
- a *Deployment view* in which the designer describes the hardware, the attibutes of interest of the target execution platform; logical distribution is also performed in this view, assigning components to logical partitions and finally physical distribution is completed by deploying logical partitions on computational nodes.

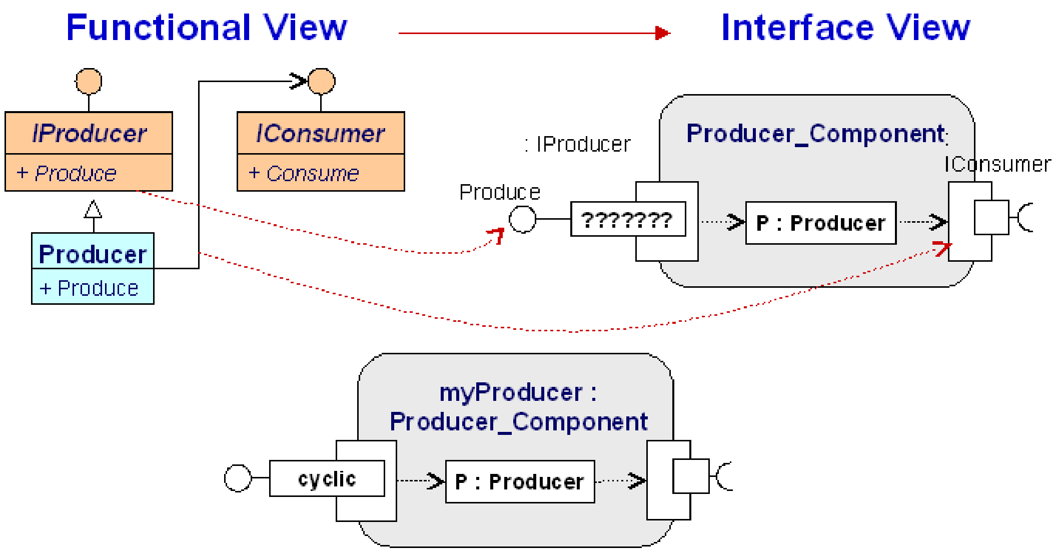

Figure 2.1: Functional containers are embedded in components in the Interface View. Concurrent semantics is later attached to provided services of the components.

The three user-level views address distinct and non-overlapping aspects of the system. Yet they are strictly and cohesively related to one another by metamodel rules that determine what semantics attach to what model elements and how. The entire design space (except for the specification of the hardware and the execution platform) is free from any concerns related to programming languages, implementation details and dependencies on the execution platform; in MDE terms we can say that the user mostly manipulates *platform-independent models* (PIM) and this is in line with the goal to raise the abstraction level of the design process to mind valued-added concerns as opposed to the low-level mechanics of implementation. An automatic model-to-model transformation is in charge of turning the PIM user model (coupled with the platform specification) into a *platform-specific model* (PSM), which contains all the details needed for its implementation on the target platform; chiefly the intended concurrent semantics that was *declaratively* specified on component services in the interface view is *implemented* in the generated PSM (which is called *Concurrency View*), according to the adopted computational model. The PSM is also a Schedulability Analysis Model (SAM) that is used to feed a tool for model-based analysis and it is the input for automatic generation of the source code for the complete concurrent architecture.

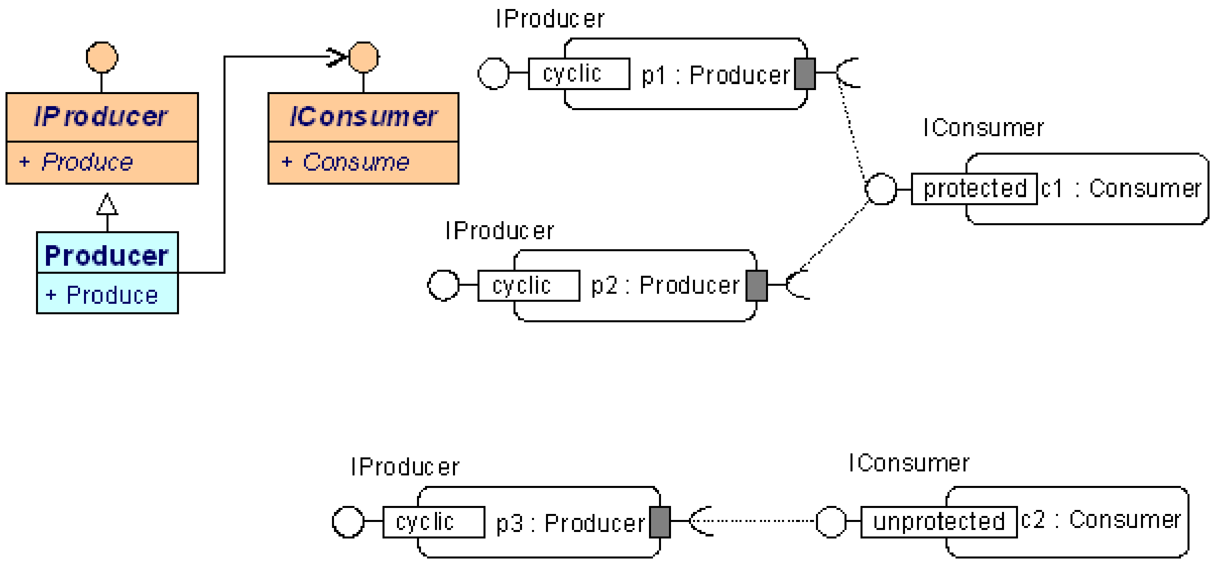

Figure 2.2: Reuse of functional specification across different concurrent architectures: on the top, operation Consume requires to be protected against concurrent access by its two clients; on the bottom, operation Consume has a single client and does not require any access protection. The functional specification is the same since the protection is provided by an additional layer applied on top of the functional specification.

Chapter 3

The Functional View

In this section we discuss the role of the Functional View in the HRT-UML/RCM methodology and reason about what kind of constructs should be excluded from it since they are not compatible with our overall intent of separation of concerns.

We want to introduce and explain the applicable constraints independently of the modeling language chosen for the functional specification. In theory the sequential behavior can be specified in any formalism of choice (UML, SDL, SCADE, Simulink, . . .) and subsequently imported in the HRT-UML/RCM infrastructure, to be embedded in components of the Interface View. In chapter 4 we discuss this import process.

As stated in earlier in this report, we want to achieve complete separation between the functional behaviour and the concurrent architecture. To attain this goal we single out the constructs which can break the sought separation of concerns since they entail non-functional considerations which should be addressed exclusively present in the Interface View. After this reasoning, we will achieve a "pure" Functional View, which is completely compatible with the HRT-UML/RCM methodology.

The first class of constructs that we investigate are those that show *concurrency-related* semantics. In this class we include constructs which: (i) cause the spawning of new threads in the system; (ii) require synchronization protocols to ensure the safe execution of operations in the face of concurrent access. Those two dimensions of concern are to be directly addressed by the concurrent architecture, and thus we exclude them from the Functional View.

The second class comprises those constructs that exhibit *time-related* semantics. Examples of language constructs that fall in this class are *sleep* (in Java), *delay* and *delay until* (in Ada), and any form of *time-out*. We thus do not include in the Functional View any reference to time or time-related suspension. The time- (and possibly space-) dependent composition of individual functional methods (or subprograms) is the concern of the Interface View (and the Deployment View, respectively). An important implication of this decision is that a system-level algorithm that has time-dependent ramifications must be described, in the Functional View, as a set of time-independent actions.

The third class of constructs is the most elusive: we do not want to use constructs that incur *blocking* semantics. In this class we include constructs that: (i) explicitly block the caller, such as for example, synchronization constructs like *join* or *barrier*, or event-related suspensions like *wait*; (ii) require an intervention from outside the application space and implicitly block the current executing thread of control. Dynamic memory allocation and dynamic creation of objects fall in the latter class of constructs and should thus be strictly prohibited.

The global enforcement of these rules means that the designer of the sequential behavior can *assume* that the specified operations are later: (i) guarded against concurrent access, if necessary; (ii) correctly attributed to a static set[1] of threads in charge of their execution. In essence, the designer is delivered of these concerns while establishing the sequential behavior of the system. (The reader should notice that a direct and important consequence of the choice of strictly separating concerns is that, strictly speaking, there can be *no* unified system concept in the Functional View, which is instead deferred to the Interface View and the Deployment View.)

The strive for what we informally called "pure" functional view earns us additional advantages. We are sure that the designer is realising a functional specification the correctness of which does not depend on timing behavior. And furthermore the functional specification, isolated from system-level concerns since its conception, it is based only on local properties and it is no more possible that we incur in the risk of polluting it with system-level properties during its design.

In HRT-UML/RCM, concurrency concerns are imposed on the model by decorating the operations exposed by components in the Interface View with the intended concurrent semantics: preserving safety from race conditions is attained by declaring the affected operations as *protected* and the automated transformation that generates the PSM protects them against concurrent access with a suitable synchronization protocol, in the form of the Priority Ceiling Protocol (PCP). The allocation of independent threads of control to system components is obtained by decorating designated methods with attributes that require that a thread of control is either periodically released by the hardware clock at a given fixed rate or activated upon software invocation or hardware interrupt. Such decorations are performed in the *Interface View*, where system components are designed and the system is created by their composition.

The reader should also notice that while constructs like timers and time-triggered events are not allowed and thus cannot be used in the *functional view*, they are effectively used in the system at run time. Their creation is under the complete responsibility of the model transformations, which can use them to implement the declarative specification of the *interface view*. Those constructs are under the control of the concurrent architecture of the system, which is automatically generated and shall not be modified by the user.

As stated before, the sequential behavior can be specified in any language of choice under the condition that all constraints imposed on the functional view are satisfied. In the following, however, we concentrate on UML [OMG07b], the preferred language for the specification of the functional behaviour in HRT-UML/RCM. The RCM metamodel conceptually mimics the basic infrastructure of UML, providing the suitable meta-classes to model classes, interfaces, properties, operations, etc.. Thus there are no conceptual (and semantic) issues in the import of a UML model. However, it is straightforward to understand that full-fledged UML violates the constraints that we are imposing on the functional view: we thus analyze what specific UML language constructs should be prohibited while using State Machines and Sequence Diagrams to specify the sequential behavior of the system.

3.1 UML State Machines

Under the constraints enumerated in chapter 3, state machines cannot be used to model any interaction between threads. Furthermore it is not possible to express time-related semantics.

A state machine is then associated to a UML class and it is solely used to model its internal behaviour.

In the following we enumerate the UML features that are prohibited in the Functional View of HRT-UML/RCM. Each feature is complemented by an excerpt of its description in the UML superstructure [OMG07b].

3.1.1 Pseudostates

The use of the following pseudostates is forbidden:

[1] Under the RCM, threads are created exclusively at system initialization time, and their total population stays fixed during the whole system life cycle (i.e., threads are never terminated).

- *fork pseudostate*: "fork vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers."
- *join pseudostate*: "join vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards or triggers."
- *junction pseudostate*: "junction vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path"
- *terminate pseudostate*: "Entering a terminate pseudostate implies that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate. Entering a terminate pseudostate is equivalent to invoking a DestroyObjectAction".

3.1.2 Events

The following event kind cannot be used to trigger a transition in a state machine:

- *SignalEvent*: "A signal event represents the receipt of an asynchronous signal instance. A signal event may, for example, cause a state machine to trigger a transition."
- *TimeEvent*: "Extends TimeEvent (of package Communications *ndr*) to be defined relative to entering the current state of the executing state machine". TimeEvent of package Communications "[..] specifies a point in time. At the specified time, the event occurs."

3.1.3 Additional constraints

In addition to the *run-to-completion*[2] semantics mandated by the UML superstructure, state machines can receive and process an incoming message (event) only if the previous message has been completely elaborated. State machines shall be in fact void of any message buffering capability (which is provided by the concurrent architecture on top of them).

3.2 UML Sequence Diagrams

UML Sequence diagram can be attributed to the behavior of a state. The entities that populate the diagram are the lifelines, which can represent properties of a UML Classifier (the pointed property is selected with the *represents* attribute). With sequence diagrams it is thus possible to model the invocation of operations on classifier properties. It should be noted however that it is only in the definition of the concurrent architecture in the *Interface View* that it is specified *how* the invocations are performed (e.g., by which thread).

An interesting feature provided by sequence diagrams is the possibility to link a specific invocation chain to a specific state and provide alternative invocation chains for alternative states.

[2]UML defines a run-to-completion semantics for event consumption, which imposes that no other event is dequeued before the processing of the previous event is fully completed. Events are consumed one by one. That means that the object will not accept new incoming events until the previous one has been fully consumed.

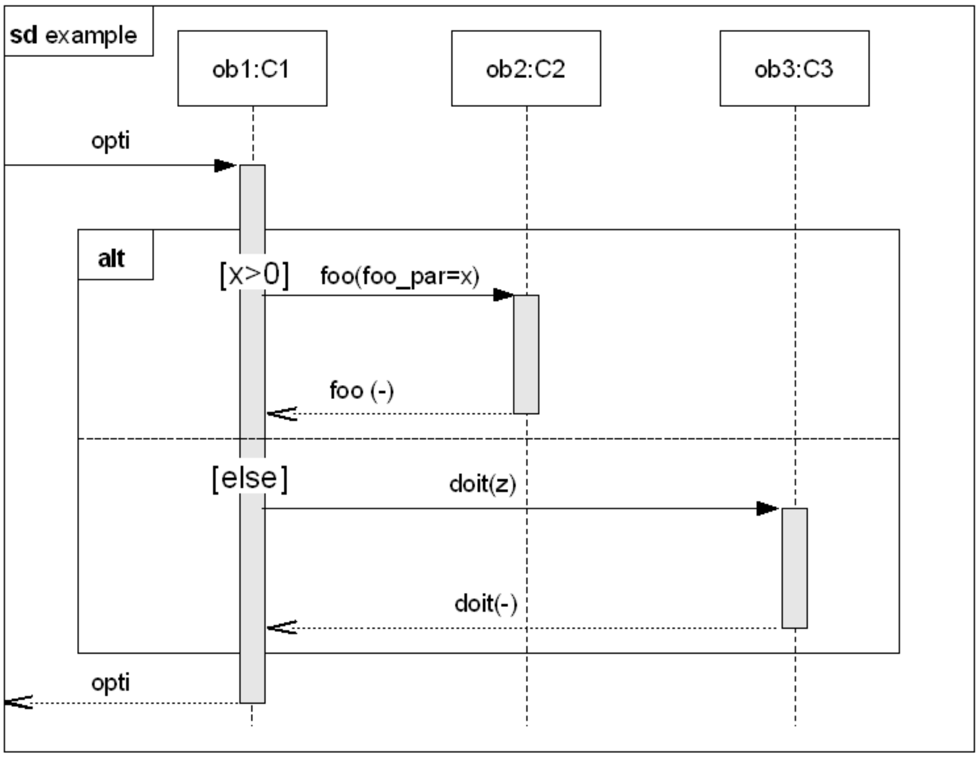

Figure 3.1: Example of sequence diagram with combined fragments

This is interesting to model intra-component execution flows precisely. That modeling can be of great advantage to dramatically improve the accuracy of model-based timing analysis, which can then be performed on single fine-grained scenarios.

In the following we enumerate the features that cannot be used in sequence diagrams:

- *CreationEvent*: "A CreationEvent models the creation of an object."
- *DestructionEvent*: "A DestructionEvent models the destruction of an object."
- *Duration*: "A duration defines a value specification that specifies the temporal distance between two time instants".
- *DurationConstraint*: "A DurationConstraint defines a Constraint that refers to a DurationInterval."
- *DurationInterval*: "A DurationInterval defines the range between two Durations."
- *DurationObservation*: "An observation is a reference to a duration during an execution. It points out the element(s) in the model to observe and whether the observations are when this model element is entered or when it is exited."
- *SignalEvent*: "A signal event represents the receipt of an asynchronous signal instance. A signal event may, for example, cause a state machine to trigger a transition."

- *TimeConstraint*: “A TimeConstraint defines a Constraint that refers to a TimeInterval.”
- *TimeEvent*: “A TimeEvent specifies a point in time. At the specified time, the event occurs.”
- *TimeExpression*: “A TimeExpression defines a value specification that represents a time value.”
- *TimeInterval*: “A TimeInterval is shown with the notation of Interval where each value specification element is a TimeExpression.”
- *TimeObservation*: “An time observation is a reference to a time instant during an execution. It points out the element in the model to observe and whether the observation is when this model element is entered or when it is exited.”

Additionally, the use of the following operators in Combined Fragments is forbidden:

- *critical*: “The interactionOperator *critical* designates that the CombinedFragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications (on those Lifelines covered by the region). This means that the region is treated atomically by the enclosing fragment when determining the set of valid traces.”
- *parallel*: “The interactionOperator *par* designates that the CombinedFragment represents a parallel merge between the behaviors of the operands. The OccurrenceSpecifications of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.”

3.3 Additional UML Constraints

- The use of Class from the Communication package is forbidden (the *isActiveAttribute* designates whether the class has its own thread of control or it is considered a passive entity).
- UML Properties shall stay fixed within a given system-wide mode of operation.

Chapter 4

Importing the Functional View

In the development of high-integrity real-time systems, it is possible that the sequential behavior is specified with different modeling languages: Matlab/Simulink for control algorithms, SDL for state machine logic, UML for the exploitation of the object-oriented paradigm, and so forth.

Describing a functional specification with a specific language of choice it is clearly an advantage, since ad-hoc tools can be used to design and verify the specification. Additionally, many of the tools can also generate source code from functional models.

When the functional specification has been completed, all functional entities must be *correctly* integrated in the *system model*. This step of integration is very critical because properties of a functional specification may greatly influence the concurrent and timing behaviour of the overall system. Furthermore, this composition activity is often performed manually, which makes it very risky and error-prone.

The integration process should satisfy the following requirements:

- it should generate a model representation of the functional specification that preserves the properties that were locally proven on it;
- each integrated functional block (component) shall not corrupt or invalidate properties verified on other functional blocks (irrespectively of the specification language used for them);
- it should be possible to draw information relevant to system-level model-based analysis (like timing analysis) from all imported functional models;
- the integration process shall provide a mean to include in the system architecture also the functional code that has already been generated by ad-hoc tools.
- data semantics of message exchanges between functional blocks implemented in distinct languages shall be preserved.

Each functional model that the designer may want to include in the system architecture shows the specific semantics prescribed by its modeling language. In MDE terms we can say that models resulting from distinct modeling languages rest on distinct metamodels.

The *Interface View* is the system view in which system components are established and functional behaviour is embedded in them to characterise their provided and required services. Importing a functional specification described in a modeling language other than that provided by the RCM metamodel means basically to import entities that express their own specific semantics.

In order to enable the creation of models with heterogeneous functional specifications it is possible to choose among two radically different paths:

1. to import the entire language semantics of each candidate language for the functional specification
2. to extract only the information of interest from the input functional model.

With the first path, we impose that the system-level language, that is the user-level language provided by the RCM metamodel, has to be expressive enough to express the semantics of all possible languages for importable functional models. This means that the RCM metamodel should be more expressive than all the corresponding functional metamodels. This is practically infeasible both from the theoretical and implementation point of view. Following the second path instead we extract from the functional model only the information required (and expressible) in an HRT-UML/RCM model.

At this point we should also verify that the functional model exhibits a semantics that does not violates the contraints specified in chapter 3. This can be done either:

1. with a set of verification rules checked by the import tool; this requires that the verification is performed during each model import and that functional models that violate the constraints are rejected;
2. designing the functional specification with a suitable subset of the functional language of choice that abides by the prescribed constraints; after establishing a mapping between the semantics of the subset of the functional language and the entities of the RCM metamodel, functional models are amenable to import by construction.

The path outlined in item 2 above was chosen in the scope of the ASSERT project for the development of the FW Profile [CEP$^+$06] by ETH, a UML Profile for the functional specification, completely compatible with the RCM methodology. A more detailed description of the import of heterogeneous functional models in RCM can be found in [BTP08].

Bibliography

[BDV03] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. *Technical Report YCS-2003-348, University of York*, 2003.

[BPV08] Matteo Bordin, Marco Panunzio, and Tullio Vardanega. Fitting Schedulability Analysis Theory into Model-Driven Engineering. In *Proc. of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[BTP08] Matteo Bordin, Thanassis Tsiodras, and Maxime Perrotin. Experience in the Integration of Heterogeneous Models in the Model-driven Engineering of High-Integrity Systems. In *Reliable Software Technologies - Ada-Europe*, 2008.

[BV07] Matteo Bordin and Tullio Vardanega. Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In *Reliable Software Technologies - Ada-Europe*, 2007.

[CEP$^+$06] V. Cechticky, M. Egli, A. Pasetti, O. Rohlik, and T. Vardanega. A UML2 Profile for Reusable and Verifiable Software Components for Real-Time Applications. In *Reuse of Off-the-Shelf Components, 9th International Conference on Software Reuse*, 2006.

[Cha06] Roderick Chapman. Correctness by Construction: a Manifesto for High Integrity Software. In *ACM International Conference Proceeding Series; Vol. 162*, 2006.

[OMG07a] OMG. *UML profile for MARTE*. 2007. `http://www.omg.org/cgi-bin/doc?ptc/2007-08-04`.

[OMG07b] OMG. *UML2 Metamodel Superstructure*. 2007. `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF`.

[PdlPH$^+$07] José A. Pulido, Juan A. de la Puente, Jérome Hugues, Matteo Bordin, and Tullio Vardanega. Ada 2005 Code Patterns for Metamodel-based Code Generation. *Ada Letters*, XXVII(2), 2007.

[PV07] Marco Panunzio and Tullio Vardanega. A Metamodel-driven Process Featuring Advanced Model-based Timing Analysis. In *Reliable Software Technologies - Ada-Europe*, 2007.

[SAE] SAE. *Architecture Analysis and Design Language*. `http://la.sei.cmu.edu/aadl/currentsite/aadlstd.html`.

[Var06] Tullio Vardanega. A Property-Preserving Reuse-Geared Approach to Model-Driven Development. In *12th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, 2006.