Composability for high-integrity real-time embedded systems

Daniela Cancila*
CEA - LISE
Center of the Nuclear Energy, France
daniela.cancila@cea.fr

Tullio Vardanega
Dipartimento di Matematica Pura e Applicata
Università degli Studi di Padova, Italy
tullio.vardanega@math.unipd.it

Roberto Passerone
Dipartimento di Ingegneria e Scienza dell'Informazione
Università degli Studi di Trento, Italy
roberto.passerone@unitn.it

Abstract

The complexity of software systems rises with the quest for better functionality and quality. The development costs that result from it may also increase significantly. Software reuse is one of the directions pursued by the engineering community to abate those costs. One of the lessons learned in the pursuit of software reuse is that composability is a central prerequisite to it. Composability ensures that the properties proven for a component in isolation continue to hold true at system integration. If component-based approaches are ever to make it into the high-integrity domain, they have to provide sufficient guarantees of composability. In this paper we discuss the role played by composability in a novel approach to model-driven engineering recently devised for use in space applications.

1 Statement of the Problem

In recent years we have witnessed a large increase in the deployment of software systems. The flexibility afforded by a software implementation, and its comparatively simpler design flow, have been key factors in this shift, and have made a range of new devices that integrate several complex functionalities possible. Software productivity, however, does not keep pace with the demand for new functions and with the time-to-market and correctness requirements of modern electronic systems. In face of this changing market landscape, the academic and industrial communities have looked at software reuse as the winning strategy to improve design productivity. Such a strategy exploits a set of architectural pre-designed components and aggregates them by appropriate composition rules meant to ensure that reuse occurs as intended. When successful, reuse can dramatically shorten the development cycle, while also ensuring high quality.

The increased adoption of software reuse methodologies is gradually shifting the cost linked to software development from the design phase to the verification and validation (V&V) phase. This occurs because components are employed in contexts that may differ from those initially intended. Thus, particular care must be taken in ensuring that the interaction between the reused components yields the correct results. The validation of this interaction is typically more complex than that of behavior alone, because of the potentially inordinate number of interaction points incurred by the execution. Numerous errors in software systems are in fact the product of faulty interactions.

Correctness by construction (CbyC) has proved effective at reducing the verification cost of software [9, 13]: in fact, the techniques based on CbyC replace the *a posteriori* validation methods [21] with a priori validation methods, in that CbyC prevents the designer from introducing errors in the early phases of a design, where they cost the most. In the literature, CbyC is defined in different ways with respect to the considered contexts; see, for instance, [9, 21, 22, 7].

In this paper we discuss the use of CbyC techniques for high-integrity real time embedded systems. In particular, we focus on the aerospace domain, and follow the modeling methodology defined in [22]. The cited methodology rests, among others, on three key pillars: *separation of concerns*, thanks to a multiple-view de-

*This work was partially supported by the IMOFIS French National Project, the ASSERT EU Project and by the University of Udine (I).

velopment style; *declarative specification*, with contractual interfaces realized via containers and automated transformations; *preservation of properties*, to assure that the contracts stipulated in the (declarative) specification are understood and actively preserved by the execution platform.

One of the pillars of CbyC is the concept of composability: the guarantee that the local properties of architectural artifacts are preserved when the architectural artifacts are aggregated or integrated together; in other words: "the whole must preserve the parts". One of the main challenges we face in this endeavor is to correctly capture and express the properties that are to be preserved upon composition.

Contribution This paper discusses properties of interest to composability and illustrates an approach to guarantee it in a Model-Driven Engineering (MDE) methodology. We discuss how model transformations must guarantee composability and show how we can achieve it.

The paper is structured as follows. Section 2 is devoted to related work. In particular we introduce the software reuse has its central mantra in all approaches which provides a component-based design where an architectural artifact is viewed as a set of components. In Section 3 we give the analysis of the problem and we introduce the case study. In Section 4 we discuss our solution of the problem and we illustrate the overall methodology by the case study. Section 5 is devoted to introduce industrial valuation. Finally, in Section 6 we discuss some open-problems and on-going work.

2 Related Work

Although there are interesting developments in the state of the art that address composability with respect to time [16, 18, 2], in this paper we focus on composability with respect to architectural features.

A wealth of methodologies, techniques and formalisms have been proposed to formalize component-based design [17, 5, 10, 3, 19, 12, 14, 22]. Most of them can be classified by the dual taxonomy of abstract vs. constructive approaches [17].

The abstract approach defines a formalism comprised of uninterpreted operations on abstract components. A component in this context is often viewed as an interface and the focus is placed on specifying the conditions under which that interface can be substituted or refined by another interface. The abstract formalism can be axiomatic, algebraic, based on graph theories as well as grammars. In [17] the authors define an axiomatic theory for (a-)synchronous models populated by agents (components) the behavior of which is defined by a set of fixed axioms. In [5] the authors use graph theory to represent components (i.e., nodes) and connection ports (i.e., edges) and to describe the composition of components and their inner structure (i.e., refinement). In [10] the authors introduce an interface algebra that supports incremental analysis of the system, caters for the dynamic introduction of new components, permits independent implementation of interfaces and simultaneous composition.

The constructive approach is dual to its abstract alternative. It centers around a given language to express components, a given platform to host and execute those components at run time, and a set of positive (shall) or negative (shan't) constraints on what those components can do in terms of run-time behavior and interaction. The constructive approach proves that some properties descend from the specification, and subsequently attempts to extend the expressive power available to the designer by constructing a set of patterns which can provably be realized by legal composition of legal components and are sufficiently general to address a large(r) set of problems in the domain of interest. This is for instance the approach taken in [12], where, with very large resonance in the software engineering community, 23 patterns were introduced and classified.

The abstract approaches have the advantage of being able to capture a large set of application domains. However, the moment a specific application needs to be addressed, the abstract theory must be instantiated to it (to permit concrete implementation) and the consistency between the implementation and the original theory must be proven, for example axiomatically. The instantiation process may thus become problematic. It may even require an inordinate amount of effort, without the guarantee of success, depending on the distance between what the theory requires and what the implementation technology actually provides.

The constructive approaches have the dual problem: they often originate from a given application domain and attempt to generalize beyond its frontier by augmenting the availed expressive power. There is no a-priori guarantee however that the generalization can actually succeed: the requirements from a given domain, normally expressed in terms of attributes and constraints, may turn out to be incompatible with those from another domain.

An intriguing contribution to the software-development challenge has recently been brought about by the model-driven engineering (MDE) movement [19] promoted by the Object Management Group (OMG) [15]. By namesake, MDE sets focus on the

use of *models* as the primary means for software construction [19, 20].

In the MDE landscape a number of distinct and non-overlapping *model spaces* are provided which allow the designer to perform software specification at multiple levels of abstraction, from higher to lower, thus progressing from conception to implementation and deployment with the help of automated transformations. Models at higher level of abstraction are intrinsically and intentionally kept independent of the target execution platform (and thus are termed "platform-independent", PIM) whereas those at lower level must resolve all dependencies on the platform (and are consequently termed "platform-specific", PSM). One of the key prerogatives of MDE is that the source code from which the executable is to be produced should be obtained from the PSM or directly from PIM.

Model transformations map elements in the PIM to elements in the PSM following model-to-model transformation rules the execution of which could and should be extensively automated. MDE does not concern itself explicitly with providing intrinsic assurances over the correctness of model transformations. The burden of which instead is shifted on those who develop the transformation rules and the automation engines, who must provide sufficient evidence (or just plain proof) of correctness. A considerable wealth of research is addressing that particular concern [19].

In MDE, *metamodel spaces* must be provided to describe the elements that may populate the models, and the allowable interrelations among those elements and the applicable constraints. A metamodel space therefore provides a higher level of abstraction with respect to the model space. In order to develop metamodel spaces the OMG provides an abstract syntax based on UML and some basic mechanisms to extend it by creating stereotypes, relationships and attributes.

3 Analysis of the Problem

We focus on using MDE for the development of high-integrity real-time embedded systems, where extreme attention is placed on the ability to deliver provable evidence of predictability (among other desired properties) and CbyC and composability are paramount concerns.

We define composability as follows. Let Γ_1 be a component such that the provided services of Γ_1 meet a given set of properties, ϕ_1. Analogously, let Γ_2 be a component such that the provided services of Γ_2 meet a given set of properties, ϕ_2. Then, a system composed by components Γ_1 and Γ_2 satisfies the composability property if it meets both sets of properties ϕ_1 and ϕ_2.

An essential feature of MDE is the separation between the space of specification and the space of the implementation. In the space of specification, a designer specifies attributes on the functionality and on the structure of model-level components. Run-time components instead reside in the space of implementation. In a CbyC-flavored approach, the run-time components must be capable of: (1) implementing the components specified in the designer's space without incurring semantic distortions; (2) executing in a property-preserving manner on a target platform.

The main contention of this paper is then the following: composability in an MDE context is achieved if and only if the population of the run-time components allowed by the implementation space are provably able to realize the components specified by the designer in the model space, guaranteeing that no semantic distortions occur during model transformation from the design space to the run-time space.

In the following we shall concentrate on properties on the behavioral structure, also known in the literature as non-functional properties.

3.1 Case study

Let us discuss about composability in the context of the fragment of a case study recently investigated in the ASSERT Project [1]. Figure 1 is a high level representation of a satellite subsystem. The TMTC component is able to receive a ground command and, according to its type, either update the current position of the satellite or send a correcting command to the propulsion system. The position component (POS) is a shared resource. The guidance, navigation and control component (GNC) is in charge of the motion of the satellite. The propulsion component (PRO) periodically controls its engine and it is also able to change its default behavior on reception of ground command by TMTC. (GNC is missing in the figure.)

For the sake of this discussion we assume that all components individually meet their requirements. For example: POS provides mutually-exclusive access to its data; TMTC component guarantees the required sporadic activation behavior. Figure 2 shows the aggregation of the components introduced above into the desired subsystem. The resulting system satisfies the composability property if the properties local to individual components still hold in the aggregation.

To constrain the implementation space we adopt the *Ravenscar Computational Model* (RCM) [23, 22]. RCM provides a modeling space that is compatible with the restrictions of the Ada Ravenscar Profile [6]. RCM addresses two crucial issues for high-integrity real-time

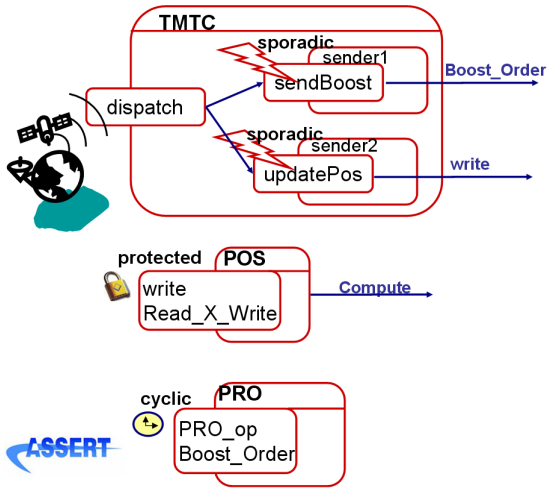

Figure 1. The components of the example system in isolation

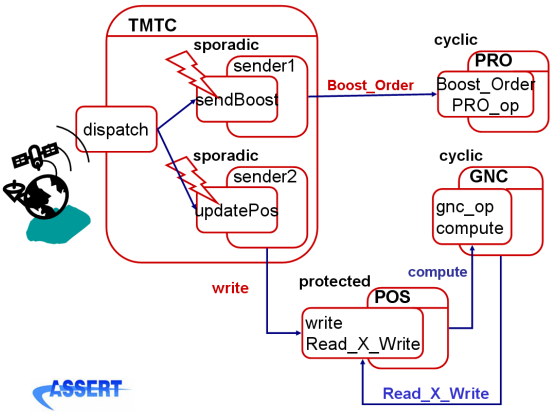

Figure 2. The assembled subsystem

systems: how to manage concurrency through trustworthy architectural choices, and how to guarantee static analyzability of a system [23].

4 Solution of the Problem

CbyC plays a central role when model transformations are to be automated [19]. In our vision, CbyC enters the MDE landscape by preventing the introduction of errors in the model spaces and in the metamodel spaces. We then proceeded as follows:

1. we impose that the PIM and the PSM shall descend from one and the same metamodel space which is adherent to RCM;

2. we formalize the model transformation from PIM to PSM by rules; we term the resulting set of rules: 'RCM Interface Grammar'.

In this paper we contend that, given any set of components in PIM, composability is achieved by proving that:

- model transformation does not introduce any semantic distortion;

- PSM components obtained by model transformation are correct representations of their PIM correspondents, and

- PSM components can be correctly realized on the target platform and can execute in a property-preserving manner.

The approach described in this paper was the theoretical foundation to the development of a prototype MDE tool environment realized in ASSERT [1] under the coordination of the European Space Agency and the participation of major European space industry, along with small and medium enterprises, tool vendors, research centers and universities across Europe. Representatives of the space industry successfully performed pilot testing of the prototype and returned very encouraging feedback on their use experience. Section 5 provides more details on the industrial experiments and their feedback.

4.1 Overall Strategy

Consider Figure 2: every method of every system component is to be decorated by a set of non-functional contracts. For example, method *PRO_op* declares a fixed-period activation requirement.

The essence of our approach to attaining composability and CbyC requires the definition of a mathematical structure that underpins the designer's model and formalizes the automatic transformation of it into the solution space, where run-time components reside.

The designer is not required to use the mathematical formalism, but the decoration of the functional methods actually uses values that emanate from the formalism. The RCM Interface Grammar is thus fully transparent to designer. Let us now delve in the technical aspects of it.

We introduce two formal languages L_I and L_C. Formal language L_I expresses the non-functional contracts on each method in the designer's space. Formal language L_C puts non functional contracts together in a way conform to RCM production rules. Furthermore each set of non-functional contracts is realized by one

run-time component correct with respect to RCM entities. (Right part of a production rule.)

More specifically, every production rule has the following form:

non terminal token in $L_C \rightarrow$ set of terminal token in L_C

that is, a PSM run-time component is fully described by a set of terminal tokens.

We introduce a non ambiguous grammar: each non terminal literal appears in only one production rule and always in the left-hand side.

A set of attributes on the functionality and on the structure decorates each literal. For example, it gives us information on visibility, deadline and protocol involved by a method. (See [7] for more details.) To take these attributes into account, we are forced to introduce a grammar with attributes. Therefore, we associate a semantic rule to each production rule.

The same set of attributes specifies a literal in L_I, i.e., it specifies a provided method in a PIM component. Some of those attributes are set by the designer. This is for instance the case of attributes on visibility, deadline or concurrent type (cyclic, sporadic, protected and passive). Other attributes are automatically generated by model transformation.

L_I is included in L_C by design, and, more specifically, tokens in L_I are a subset of non terminal tokens in L_C. Therefore, given a token in L_I we can always find one production rule (because the grammar is non ambiguous) and then a PSM run-time component which is able to realize the method specified by the designer in the PIM space.

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 non-ambiguity of the RCM Interface Grammar warrant that the ensuing model transformations incur no semantic distortion.

We discuss our claims by returning to the case study introduced in section 3.1.

4.2 Application to the case study

In the following we illustrate how our strategy applies to the PRO and TMTC components respectively, and then, in section 4.2.3, we discuss two specific situations where composability issues arise.

In section 4.2.1 we see how two methods provided by the PRO component (PRO_op and Boost_Order) are realized by one and the same run-time component. This is a significant case because it involves one of the most complex rules in the RCM Interface Grammar.

In section 4.2.2 instead we see that three distinct run-time components are needed to realize the TMTC component, as a result of the application of two other grammar rules.

(Owing to space limits, we only discuss production rules without giving all the corresponding semantic rules. Overall, the RCM Interface Grammar includes ten rules.)

4.2.1 PRO component

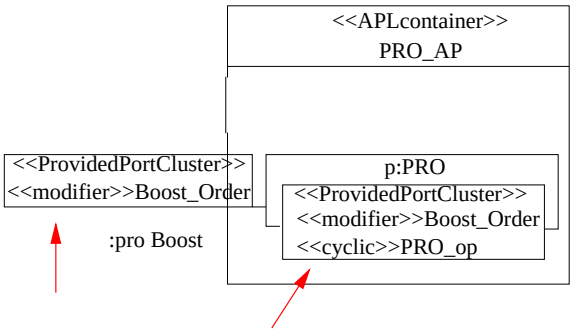

Figure 3. PRO component in the PIM space

Figure 3 shows the PRO component in the PIM space. The designer specifies the concurrent behaviour of method *Boost_Order* by setting a modifier attribute on the corresponding method. Moreover, she labels the method by adding temporal constraints (e.g., deadline). Once the 'modifier' is set, a list of attributes are also automatically set. (A preliminary draft is given in [7]). The Modifier corresponds to terminal token $PAER$ in L_I. Since L_I is a subset of L_C, we look at the production rules. Since the grammar is non ambiguous, only one rule can be selected. The choice depends on syntax (PAER) and semantics (the list of attributes). Then we have:

cyclic run-time component with modifiers
$PAER \rightarrow (ASATC_c, CER, PSER)$

In the rule, $ASATC_c, CER, PSER$ are terminal tokens. Rule states that a modifier ($ASATC$) can be introduced only if the cyclic default operation (CER) has been already specified. Therefore, the designer needs to set also the cyclic default operation (PRO_op). The provided methods by PRO are realized by only one run-time component and more precisely by a cyclic run-time component with modifiers. $PSER$ expresses a state-dependent condition controlled by a synchronization protocol. $PSER$ is automatically added when a CER attribute is set. Modifiers and the default cyclic operation use the same $PSER$.

4.2.2 TMTC component

Figure 4 represents the TMTC component. It provides three methods: dispatch, sendBoost and updatePos. Method dispatch is a protected resource and it is able to receive ground commands. If the ground command specifies the new position that the satellite must take, then updatePOS is activated. If the ground command is directed to the propulsion component, then sendBoost is activated. updatePos and sendBoost are both methods with sporadic activation. Both have private visibility, that is only method dispatch is authorized to invoke them.

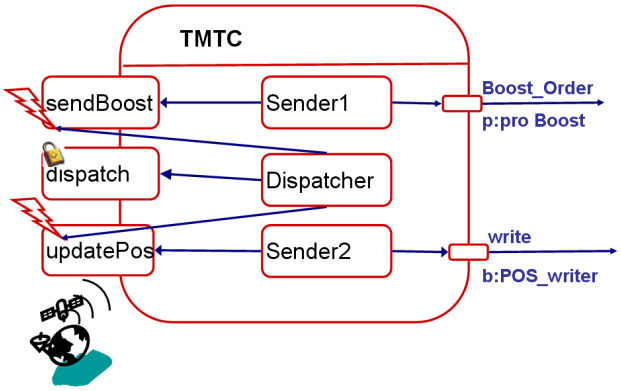

Figure 4. TMTC component in the PIM space

Dispatch corresponds to literal $PAER^+$ in L_I. When the designer sets dispatch to protected, some attributes are automatically set on the method. For example, the type of protocol which warrants mutual exclusion on access to data. Since L_I is included in L_C and the grammar is non ambiguous, only one rule can be applied to dispatch.

protected run-time component
$PAER^+ \rightarrow (PAER_{st}) \mid (PAER_{st}, PAER^+)$

On the left-hand side of the production rule, $PAER^+$ is a terminal token in L_I and non terminal token in L_C and represents the dispatch method. $PAER^+$ can generate either a protected PSM component which realizes only one method ($PAER_{st}$) or a protected PSM component which is able to realize a finite list of methods. The production rule is related to a semantic rule which formalize the following statement: a protected component can realize a finite list of $PAER^+$ only if they share the same local resources (functional state). We recall that resources in RCM are statically fixed a priori and no resource can be create at run-time. For this reason, the list of methods is always finite.

Method sendBoost corresponds to literal PAER in L_I. PAER is a terminal token in L_I and non terminal token in L_C. Following the strategy described above, we select the following production rule:

sporadic run-time component
$PAER \rightarrow (START, PSER)$

Method sendBoost is realized by a sporadic component. Its attribute are inherited by the START literal (semantic rule). PSER is a guarded entry which is opened by the invocation of START. Method updatePos is treated analogously to method Sender1.

Unlike the PRO component, the TMTC component is realized by three PSM components: one protected, and two sporadic. As a result of the application of these rules, model transformations do not introduce any semantic distortion and consequently the resulting set of PSM components is a correct representation of TMTC component.

4.2.3 Composability

The realization of the TMTC component provides an example of how composability is achieved: the designer specifies methods in the TMTC component and interconnects them (sendBoost and updatePos are invoked by dispacth) with the guarantee that the attributes on set on the individual methods are not failed during model transformations. For example, attribute "protected" on method dispatch is not affected by its interaction with sendBoost and attribute "deadline" on sendBoost is likewise not affected by the interaction with dispatch. Model transformation acts on individual methods so that composability is guaranteed when they are integrated together.

Another example of composability is the following. By functional requirements, an invocation by the TMTC component may change the operational behavior of the propulsion component (PRO). We have shown that TMTC and PRO preserve their own attributes upon model transformation. When we integrate TMTC and PRO components in a single system (see Figure 2), we are guaranteed that each of them preserves their own original properties on composition. The key is that the transformation rules we have seen operate on individual methods in isolation. Therefore, TMTC methods dispatch, sendBoost and updatePos as well as PRO methods Bootst_Order and PRO_op are all correctly realized by fitting and appropriate PSM components as before integration. Attributes specified on methods in PIM are inherited by methods in PSM and then realized by PSM components with no semantic distortion (because $L_I \subset L_C$ and the non ambiguity of the interface grammar). Model transformations applied to TMTC and PRO components enable them to

continue to meet the original properties when TMTC and PRO are integrated together.

5 Current Status

To assess the viability and effectiveness of the MDE approach outlined in this paper, a team based at the University of Padua (UPD) developed a concept demonstrator 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 instalments.

GMF, ATL and MOFscript [11] 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; 8,000 lines of MOFscript to implement code generation; 7,500 lines of Java to complete the graphical editor (in addition to 150,000 lines generated automatically by GMF). See reference [8] for wink clips on how our tools operate.

The demonstrator was used by two industrial partners to re-design sizeable subsets of internal reference on-board systems of theirs. The essential requirements that the industrial teams placed on the case studies included:

- active enforcement of separation of concerns: the user 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

- proof of correctness of the automated transformations that turn the user model (the PIM) into the run-time component space (the PSM) 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

- expressiveness and coverage of the non-functional (including, of course, real-time) requirements settable on the contractual interfaces

- ability to explore the solution space in the PSM in a round-trip feedback-based manner originating from the user model space at PIM. The interested reader is referred to [4] for details on the forms of static real-time analysis supported in the proposed approach.

Both experiments earned us authoritative confirmation that our vision addresses the industrial requirements outlined above and that it does meet some of them fully. (i) Separation of concerns in a PIM-centric user space was found to be both desirable and achievable. (ii) The declarative and platform-independent specification permitted in the PIM space has potential for decreasing the development time considerably by sparing the burden of decomposing the system down to primitive run-time entities and then having to prove their local and global correctness. (iii) The provision of proof support for transformations and verification, in the form, for instance, of the RCM Interface Grammar 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.

6 Conclusions and Outlook

The need to assemble increasingly critical and complex functionalities in high-integrity production systems requires developers to attain more aggressive levels of software reuse as well as unprecedented levels of automated code generation.

Composability and CbyC are essential features in techniques oriented to software reuse. Their importance is further increased in the high-integrity domain where extreme attention is placed on the ability to deliver provable evidence of the satisfaction of the required properties.

In this paper we have discussed a strategy to ensure composability and CbyC in an MDE approach. PIM and PSM components are described by two languages, which share syntax and semantics. By design, the language for PIM components (named L_I) is included in the language for PSM components (named L_C). This is possible because we impose that PIM and PSM shall descend from one and the same metamodel.

Resting on a single metamodel however is not practical for the development of heterogeneous systems, which rather require the use of multiple metamodels. Our current work explores the problem space in this particular direction.

References

[1] http://www.assert-project.net.

[2] E. Bini and G. Lipari. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 2005.

[3] S. Bliudze and J. Sifakis. The Algebra of Connectors - Structuring Interaction in BIP. In *Int. Conf. EMSOFT*, pages 11–20, 2007.

[4] M. Bordin, M. Panunzio, and T. Vardanega. Fitting Schedulability Analysis Theory into Model-Driven Engineering. In *Euromicro Conference on Real-Time Systems (ECRTS 08)*, pages 135–144. IEEE, July 2008.

[5] R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Style Based Reconfigurations of Software Architectures. Technical Report TR-07-17, Università di Pisa, 2007.

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

[7] D. Cancila and R. Passerone. Functional and Structural Properties in the model driven engineering. In *Int. Conf. ETFA*, IEEE, 2008.

[8] D. Cancila, M. Trevisan, and T. Vardanega. A gentle introduction to the HRT-UML/RCM methodology. http://www.math.unipd.it/~tullio/Research/ASSERT/Tutorial.

[9] R. Chapman. Correctness by Construction: A Manifesto for High-Integrity Software. volume 162 of *ACM Int. Conf.Proc. Series*, 2006.

[10] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental Schedulability Analysis of Hierarchical Real-Time Components. In *Proc. EMSOFT 2006*, pages 272–281, 2006.

[11] http://www.eclipse.org/modeling.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[13] A. Hall. Realising the Benefits of Formal Methods. *Journal of Universal Computer Science*, 13(5):669–678, 2007.

[14] M. Kircher and M. Volter. Software Patterns. IEEE Software, July-August 2007.

[15] OMG. http://www.omg.org/.

[16] J. Pulido, S. Uruena, J. Zamarro, T. Vardanega, and J. D. L. Puente. Hierarchical Scheduling with Ada 2005. In *11 Ada Europe Int. Conf. LNCS*, 2006.

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

[18] A. Sangiovanni-Vincetelli and M. D. Natale. Embedded system design for automative applications. *Computer, IEEE Computer Society*, 2007.

[19] D. Schmidt. Model-driven engineering. *IEEE Computer*, pages 25–31, February 2006.

[20] B. Selic. From Model-Driven Development to Model-Driven Engineering. Keynote talk at ECRTS'07. http://feanor.sssup.it/ecrts07/keynotes/k1-selic.pdf.

[21] J. Sifakis. Embedded Systems - Challenges and Work Directions. In LNCS, editor, *Principles of Distributed Systems*, volume 3544, 2005.

[22] T. Vardanega. A Property-Preserving Reuse-Geared Approach to Model-Driven Development. In 12^{th} *IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 223–230. IEEE, August 2006.

[23] T. Vardanega, J. Zamorano, and J. de la Puente. On the Dynamic Semantics and the Timing Behaviour of Ravenscar Kernels. *In Real-Time Systems, Springer-Science*, 29:58–89, 2005.