A Domain-specific Metamodel for Reusable Object-Oriented High-Integrity Components

Matteo Bordin and Tullio Vardanega

University of Padua, Department of Pure and Applied Mathematics,
via Trieste 63, 35121 Padua, Italy
{mbordin,tullio.vardanega}@math.unipd.it

Abstract. Owing to their extremely high verification and validation costs, high-integrity systems tend to resist the introduction of innovative software technologies: as a notable example, the object-oriented paradigm has been so far dispensed with entirely, mostly on account of the difficulty and cost of performing static verification in the face of (multiple) inheritance, polymorphism and dynamic binding. Domain-specific verification constraints have thus prevented the high-integrity community from harvesting the potential benefit that should ensue from the application of object-oriented design, for example the ability to adaptively reuse pre-existing components via inheritance and overriding. In this paper we show how a highly constrained domain-specific metamodel can rise the abstraction level of the verification and validation process phase and consequently allow model-based static analysis of object-oriented high-integrity systems that use controlled forms of inheritance, polymorphism and dynamic binding.

Key words: Domain-Specific Modeling, High-Integrity Systems, Object-Oriented, Software Reuse

1 Introduction

One of the advantages that the object-oriented paradigm exhibits over classical imperative programming is the ability to devise sophisticated forms of reuse and adaptation using inheritance and overriding. A component can extend a pre-existing one (via *inheritance*) and possibly provide a new implementation of parts of the inherited functionalities (via *overriding*); furthermore, a component can express its functional obligations and capabilities in terms of (abstract) interfaces, and thus be reused in multiple contexts by providing alternate implementations of the required interfaces (via *polymorphism*). Software frameworks constitute an especially attractive example of the application of object-oriented reuse strategies. Reusable object-oriented software frameworks [1] require the design of a cohesive set of components that communicate through abstract interfaces: while the framework implements the invariant part of the system instantiation, application-specific behavior is introduced by means of inheritance and overriding. More advanced and disciplined adaptation and reuse strategies that use the object-oriented paradigm can be attained by employing more articulate formal specification tools and techniques, for example Design-by-Contract [2].

We contend that advanced reuse strategies such as those typical of the object-oriented paradigm would be especially useful for systems with high verification and validation costs, which must confront with stringent engineering standards: the disciplined reuse of an already verified component should arguably require less effort than the development of a brand new component that provides similar functionalities. Among the families of systems which are exposed to high verification costs and would therefore benefit from advanced reuse strategies, we can surely include high-integrity systems, notable examples of which are on-board control systems for aerospace and automotive applications as well as air traffic management systems.

In the high-integrity domain, the whole development process strives to produce systems which would be easy to validate [3]: the verification, validation and (where applicable) certification processes often absorb as much as 2/3 of the overall development costs. For this reason, best-practice principles loathe technologies which may be feared to increase those costs. Reuse strategies based on inheritance and overriding usually require the use of dynamic binding to resolve invocations at run time because not all types (but only their implemented interfaces) are known when a reusable component is designed. The use of dynamic binding has two main repercussions on the verification of a high-integrity system: (i) given an invocation it is not possible to statically determine the type of the object that resolves it; and (ii) the number of possible execution paths caused by each dynamically-bound invocation is equal to the number of (sub)types of the type of the invocation object. As a result, the cost of statically providing *full* source-level coverage of all possible execution paths (and not just methods!) as required by certification standards such as the DO-178B [4], is simply prohibitive for systems that use dynamic binding [5].

The verification cost of a high-integrity system which uses dynamic binding being so frightfully high, semantic mechanisms such as inheritance, polymorphism and overriding are as good as banned from the domain of interest: as a consequence, current high-integrity systems cannot really ripe any benefit from advanced reuse and adaptation strategies as permitted by the object-oriented paradigm, for as mature as they presently are. The idea we discuss in this paper is to apply a model-driven approach [6] to the development of high-integrity systems and use the *model* (as opposed to the *source code*) as the object of verification and validation. Models are currently not considered a viable alternative to source code for verification purpose, mostly because all too often they are feared not to represent a faithful and precise description of the system as it will actually be at run time. Of course a fully model-based approach requires the introduction of an additional compilation step (from model to source) which in turn needs to be verified to assure that the semantics expressed at the higher level (the model) is preserved flawlessly at the lower lever (the code). To the best of our knowledge, the only industrial-quality exceptions to the scarce take-up of model-based approaches in the high-integrity domain are SCADE [7] — a formal textual language and an attached tool suite which support advanced static analysis to prove properties of state machines and concurrent execution — and the Matlab tools (for example Simulink [8]). They however don't fully support object-orientation or adaptive reuse and are quite limited in scope, as they lend themselves to only a subset of the functionalities typically exhibited by high-integrity systems: SCADE for example promotes synchronous programming only and only supports state machines modeling. More recent approaches, which for example entail the use of AADL [9] as a source for model-based timing analysis [10], also suffer from limited (or rather non-existent) attention to functional modeling geared to adaptive reuse. We contend that, by strictly adhering to a domain-specific metamodel, we can force the design semantics to be precise and constrained enough to assure that the model itself contains all of the information required to affordably verify high-integrity systems which use dynamic binding.

Using the modeling language described in this paper it is thus possible to develop and reuse high-integrity components that can be adapted through inheritance and overriding, while still allowing a feasible verification process compliant with the most stringent industrial standards. Finally, by proving our thesis, we contend that domain-specific modeling can not only be used *constructively* — to improve the effectiveness of the design and implementation process — but also to constrain the design semantics so as to ease the verification and validation process.

1.1 Related work

The use of object-oriented semantics for high-integrity real-time systems has long been debated as a possible avenue to decrease the cost and improve the effectiveness of the development process. Nevertheless, the extent of object-oriented semantics that is deployed in practice is most often deprived of dynamic binding and polymorphism, thereby shrinking to inheritance only: this is for example the case with the SPARK programming language [11]. Other approaches [5] do allow a controlled use of dynamic binding: if all types are known at compile time, the code that issues a dynamically-bound invocation can be substituted by a switch statement which encompasses all possible (sub)types of the invocation object. A suitable compilation system can automatically perform the source substitution just before entering the second pass of compilation: in this manner, once the actual type of the invocation object is determined at run time (through the equivalent of `instanceof` in Java) it can be downcast and the dynamically-bound invocation can be substituted by a statically-bound one.

Unfortunately, source-based substitution approaches suffer from two main limitations: (i) since certification standards such as DO-178B [4] require full source-to-executable traceability, the code substitution adds an extra transformation step that must be verified, thereby further increasing the complexity of the compiler which must itself be certified, and traced, which also increases the verification cost; (ii) a switch statement just enumerates all possible invocations and requires coverage of all possible execution paths: as noted in [5], if the number of "dispatching" methods is `n` and the number of dynamically bound calls is `d`, the number of possible execution paths is O(`d`×`n`). Source-based substitution approaches for the verification of systems that use dynamic binding thus are not cost effective in the high-integrity domain: this observation is in fact the main grounds on which we argue for the use of a formal representation of the system, the model, as the object of the verification phase.

With a view to covering as much of the development segment of the life cycle of high-integrity systems as possible, we designed a domain-specific metamodel which supports both functional and non-functional modeling: functional modeling addresses concerns functional requirements only (that is, the sequential specification of the system); non-functional modeling addresses non-functional requirements (concurrency, liveness, safety and reliability). We called the metamodel *RCM* [12], shorthand for **R**avenscar **C**omputational **M**odel [13], which is the concurrency model enforced by the RCM metamodel. While initially conceived for the aerospace domain, the RCM metamodel can arguably be used for other families of high-integrity systems.

The RCM metamodel is conceptually traceable to a UML2 profile [14]: for practical reasons however we chose to extend the UML2 semantics with domain-specific notions (which made us deflect from a canonical profile) and equip it with the information required to support static analysis of the system. The approach we followed in designing the RCM metamodel was pretty much bottom-up: we started off from an abstract computational model rigidly constrained to ensure that important properties of the system could be statically proven and also preserved at run time; and then, around this core semantics, we developed a high-level modeling language for the designer to use. The Ravenscar Computational Model [13] which the RCM metamodel is based upon, directly emanates from the Ravenscar profile for the Ada programming language [15], which defines a set of semantic restrictions to guarantee deterministic execution and static timing analyzability for concurrent, priority-based systems. The Ravenscar Computational Model abstracts the Ravenscar profile away from a particular programming language and makes it language-neutral; while it expresses its semantics using concepts common to real-time systems — tasks, shared resources, event handlers and so forth — the RCM metamodel acts as sort of abstraction layer for the low-level semantics that is needed to express those entities and constructs in any concrete implementation language: one of its main credits is to permit a high-level, sound, effective and object-oriented modeling of Ravenscar-compliant systems (see figure 1 and reference [12]). To achieve precise model-based analysis, an RCM model (a *platform-independent model* or PIM) coupled with the *platform specification* - the target execution platform and the run-time kernel specification - is transformed into a semantically-equivalent model placed at a lower abstraction level (that is, more concrete), which represents the same system deployed on a specific platform (which is thus called *platform-specific model* or PSM); the platform specification includes the cost of run-time primitives as specified in [16], for example: context switch; moving a task to the ready queue; acquiring or releasing a lock. Tasks, shared resources and event handlers thus emerge during the transformation process, along with all the information required to evaluate their timing behaviour in a concurrent environment (see figure 1). Assuming that the PSM and the source code representation are semantically equivalent (which should be the case if the code generator was proven correct) the PSM can be considered as a true blueprint of the executable system at run time: both representations express one and the same semantics and the former is augmented with all the attribute information that describes the run-time platform. Interestingly the static nature of the model is not a limit to verification, as its Ravenscar compliance guarantees that its execution behavior can be statically verified in a trustworthy manner.

Following our proposed strategy, we arguably overcome one of the main limitations of model-driven technologies when applied to high-integrity systems, that is, we maintain a tolerable distance between the model and the executable system without causing the designer to lose control over the run-time consequences of their specifications. For this reason we can thus fruitfully use the PSM as the pivot element for model-based analysis (see figure 1 and reference [17]).

The RCM metamodel also comes with a model-to-code transformation engine that generates source code from the PSM in full compliance with the Ravenscar constraints [18]: our primary target language at present is Ada 2005 [19], while Real-Time Java [20] is in pipeline. We currently achieve 100% code generation for the concurrent architecture and object modeling (component instances and their provided/required interface bindings); the code generation for the functional specification is presently limited to the generation of the state machine transition logic, but an action language can be used to fill the action semantics [21]. The RCM metamodel and the associated tools have been developed with Eclipse technologies, in particular EMF, ATL and MOFscript [22]: the overall transformation process and PIM/PSM semantics are fully described in [12].

In this paper we concentrate on how the PIM design can make use of object-oriented semantics and adaptive reuse through inheritance and overriding while still guaranteeing static analyzability.

2 Object-oriented modeling with the RCM metamodel

The RCM metamodel provides semantics for functional modeling via UML2 constructs that essentially range classes and state machines [21]; we further allow the designer to decorate component *ports* using a declarative language to specify the concurrent semantics required of each provided method [12]. The method profile and functional requirements of a class univocally determine the provided and required interfaces of a component. Component instances are then interconnected by binding the required ports of an instance with the provided port of another instance; each instance is eventually deployed on a given computational node. The RCM metamodel strictly enforces constraints on the modeling of both types (components) and objects with the specific goal to assure that the model can be safely used as the object of system verification. In the following we discuss the most important constraints imposed by the RCM metamodel in so far as they concern object-oriented semantics; we do so in a bottom-up fashion, starting from object modeling.

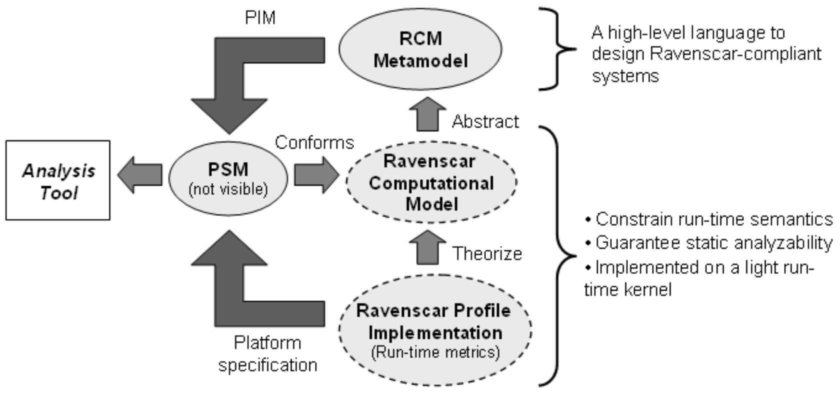
PIM
RCM Metamodel
A high-level language to design Ravenscar-compliant systems
Abstract
Conforms
Analysis Tool
PSM (not visible)
Ravenscar Computational Model
• Constrain run-time semantics
• Guarantee static analyzability
• Implemented on a light run-time kernel
Theorize
Ravenscar Profile Implementation (Run-time metrics)
Platform specification

Fig. 1. The RCM metamodel, the Ravenscar Computational Model and the Ravenscar profile: model-based analysis is possible thanks to the close relation between modeling and run-time semantics; the PSM is automatically generated form the PIM and the platform specification; the PSM is not visible to the user to avoid any direct contact between the designer and the lower-level specification.

2.1 Constraints on object modeling

Object modeling is a fundamental step to system design in that the execution flow of the system can be determined only at instance level, that is in an object diagram.

The main entities of object modeling are component instances. Following the UML2 inheritance of the RCM metamodel, each component instance is characterized by a set of *slots*, each one representing a *property* of the component itself: we then have a slot for each *provided* and *required port* of the component. Unlike UML2 [23], though, the RCM metamodel requires a very precise discipline of object modeling: for this reason, the metamodel only accepts component instances with exactly one slot for each provided/required port. In this manner, we avoid any omission from the designer: for example, we oblige them to *always* trace the link between the slot of a required port and the slot of a provided port; and we *always* force the explicit specification of the value of class features, for example the criticality of the service behind the slot of a provided port. The required constraints can be easily implemented by the model editor, which can automatically generate all required slots every time an object is designed.

To bind a required port to a provided one, we trace a *link* between them: the link is basically a reference of the component instance that requires a service to the component instance that provides it. To permit precise static analysis of the execution flow, we must make sure that the link between a required and a provided port is fixed once and for all during the initialization of the system: even if the *run-time* mechanism to invoke a service through a required port is dynamic binding, looking at the model we can statically and with certainty determine which precise component instance resolves that particular invocation (cf. figure 2). We force the link to be permanent for the entire lifetime of the system for a further important reason: by doing so we have just one possible execution flow that exits a required port and enters a provided one. During the verification phase, we can thus regard that flow as the sole possible resolution for the invocation through a required port, thereby discounting all alternatives as impossible: in this manner the analysis is vastly simplified. It is important to notice that a static and permanent binding between a required and a provided port is a common feature in high-integrity systems (cf. e.g.: HRT-HOOD [24], which however is very timid with the use of object-oriented semantics).

To fully determine the execution path *across* several components instances it is also necessary to know which required port is traversed in response to the invocation of the service behind a given provided port of the *same* component instance: in other words, it is necessary to know which required services a provided service invokes during its execution. This information is automatically extracted from class modeling, as discussed in section 2.2 below.

The *static* extraction of the execution path is a fundamental step toward model-based analysis. The execution path extracted at PIM level is of course still valid at PSM level, because nothing in the transformation logic we apply ever occurs which may alter it. Even in presence of dynamic binding, it is possible to statically determine the execution path *of each task* which appear at PSM level to reflect the PIM view, in full account of their concurrent action semantics, for example the acquisition/release of locks or their suspension periods: this information, coupled with the platform specification, is used to appropriately instruct the analysis tool to evaluate the timing behavior and performance of the system.

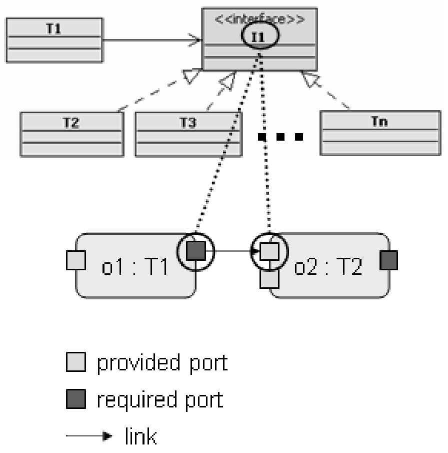

Fig. 2. Execution path in an object diagram: the invocations use the dynamic binding mechanism, but the model statically shows which object (and its type) resolves the invocation.

2.2 Constraints on component modeling

While object modeling in the RCM metamodel allows one to statically determine the execution path *across* different components instances, it is necessary to assure component (class) modeling provides all the information needed to automatically perform the required extent of analysis and transformation on it (that is, without involving user intervention past the modeling stage).

One key element of our strategy to warrant that the system be statically analyzable is to force the designer to be extremely rigorous and accurate at model level with the intent of making the model a most trustworthy representation of the actual run-time system. To achieve this goal, the designer must specify a full functional contract for each concrete method of a class: the contract includes the accessed class members and the methods invoked during execution. Figure 3 depicts a simplified graphical representation of a relevant part of the RCM metamodel: an RCMoperation must specify the accessed class members (the *worksOn* relation pointing to the metaclass Property) and the invoked operations with optional specification of the upper bound on the number of invocations (the RequiredOperation metaclass). The methods invoked during execution are expressed in terms of invocation objects and invoked methods. Arguably, the level of detail we require of the designer is not that uncommon in the high-integrity domain, as it can be well testified by the rigid semantic requirements promulgated by programming languages such as SPARK [11] and architectural design languages such as AADL [9].

The topology and expressivity of state machines is constrained as described in [21]. A state machine cannot contain any concurrency or time-related semantics, so as to enforce strict separation of concerns between functional and non-functional modeling; the permissible structure of state machine is expressly devised to facilitate verification, for example by constraining the allowable number of entry points or the number of transitions entering a state.

When a class is regarded as a component, each method is provided through a provided port; similarly, each operation invoked by a method contributes to the required ports of the component itself. Since the designer is forced to specify the operations invoked by each method, it is possible to automatically extract the "path" from a provided port to a required one: for example, if method `p()` invokes method `r()` on object `o`, then there is a path from the port providing `p()` to the port requiring `o.r()`. This simple example shows how from a functional contract it is possible to extract the component-level inner execution path from a provided port to a required one: the RCM metamodel enforces complete semantic equivalence between the operations invoked by a method and the required ports of the corresponding component.

Since the path from a provided port to a required port is class-invariant (i.e., it is valid for every objects of that class) it is possible to use this information to automatically augment the information contents of the object diagram and thus determine the execution path across provided and required ports of component instances.

One further important constraint holds on the choices of the invocation object for a required operation (or required port, since we have just shown that they are semantically equivalent). In section 2.1 we explained that a link between a required port and a provided port of two component instances must be permanent to statically determine the execution path once and for all: that link is semantically equivalent to the object target of the invocation through a required interface (object `o` in the previous example). To assure that the link is statically fixed it is necessary to fix the object target of the invocation once and for all: for this reason the object target of the invocation must be a property of the component that requires its services (in other words, a class member) and conversely cannot be a parameter of the method which invokes the requiring port. In UML2 terms, we would say that the functional dependencies of a class are determined by its properties only, and not by the parameters of its methods. The property of a class can indeed be forced to always reference the same object, while a method parameter can possibly reference distinct objects at distinct invocations. To assure that the object target of a required port and referenced by a property is statically fixed once and for all, it is sufficient to hide it in the private part of the class and force its "setter" (the method which changes the object referenced by a class property) to be called just once during the initialization phase (cf. section 2.3).

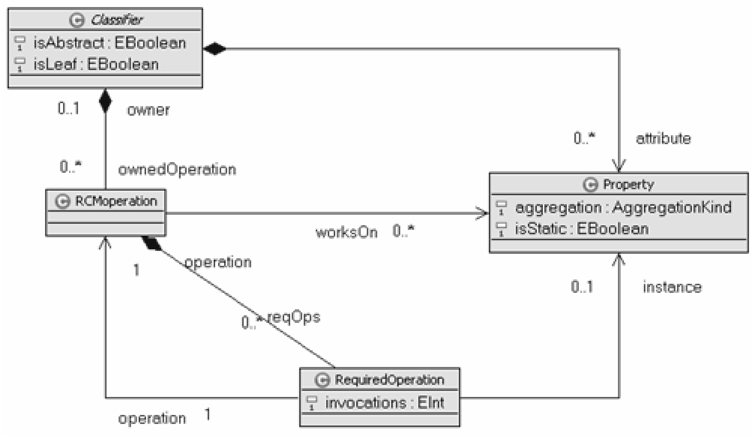

Fig. 3. RCM metamodel: an operation must specify its functional contract (accessed class members and invoked operations).

2.3 Constraints on action semantics

It is of course necessary to assure that the execution of the system doesn't violate the constraints imposed by the RCM metamodel on class and object modeling. Some of the constraints enforced on action semantics regard memory management: since high-integrity systems usually require thorough static analysis of memory usage, dynamic memory allocation is strictly prohibited. However, the code generation engine or the target programming language can provide a way to statically pre-allocate fixed-sized heap pools: this is for example the case with the Real-Time Specification for Java [20] as well as with Ada 2005. In addition, dynamic class loading is prohibited, as it would render any form of static analysis totally hopeless: if some types were not known at compile time, it would obviously be impossible to statically determine how to resolve a dynamically-bound invocation even in the presence of the constraints enforced by the RCM metamodel.

Filling the action of the state machine may potentially violate the functional contract attached to individual provided services: for example, if the contract behind method `p()` specifies that it can *only* invoke method `r()` on object `o`, then the actions of the state entered by the transition triggered by `p()` may only contain the `o.r()` invocation. We are currently exploring ways to actively constrain the semantics allowable for the designer to express state machine actions (cf. section 3).

Finally, to assure that the binding between a required and a provided port is statically fixed and permanent, links between ports of component instances can be created just once at initialization phase. The action of creating a link is semantically equivalent to executing a setter method (cf. section 2.2): the constraint in question thus has a direct reflection on the action semantics of the metamodel/action language and basically amounts to stipulating that setters can be executed only once during the initialization phase.

3 Current results and future research

The main goal of this paper is to show that we can achieve full model-based static verification of high-integrity systems while using dynamic binding to resolve invocations at code level: by proving this contention, we permit the deployment of reuse and adaptation strategies which exploit inheritance and overriding in the very domain of high-integrity systems.

The critical assumption on which the whole of our contention and the viability of our proposal rest requires that the model be a trustworthy representation of the real system at run time. A direct implication of this requirement is that neither the generated nor any hand-written code intermingled with it (if any) may violate the contracts specified in the model. Proving the truth of this assertion is quite easy for generated code (for it suffices to validate the generator engine once) but it may unfortunately require considerable effort for handcrafted code manually inserted through action languages: to address this problem area, we are presently developing a prototype tool (using the `xText` framework [25]) which allows us to automatically validate the hand-written code placed inside state machines *against* the model specification. We however do not plan to support the modification of the model via hand-written code as this choice would violate the design-by-contract principle.

So long as the model is a trustworthy representation of the system at run time the static determination of all possible execution paths from the model is quite straightforward: the functional contract behind each method lets us infer which required port is traversed in response to the invocation of a provided service; and the static nature of the binding between required and provided ports of component instances lets us determine the object target of the invocation even in the presence of dynamic binding.

To evaluate the effectiveness of our approach we developed two prototypes, both of which are part of the on-board software of a real industrial satellite: the first prototype is a battery management system, which has been designed from scratch; the second is a telecommand and telemetry management system, which has been designed by adaptation of pre-existing components from an object-oriented framework which makes extensive use of dynamic binding [21].

The degree of precision required by the RCM metamodel has proven to be in line with current industrial practice for high-integrity systems in the aerospace domain. Furthermore, most of the specification (accessed variables, functional dependencies) is automatically (and formally) contained in the model itself, which can be then used as an additional aid to the automated production of system documentation.

The automated extraction of all possible execution paths had proven useful to statically verify the prototypes. For example, by providing the worst-case execution time of individual methods (determined with the assistance of tools like Bound-T [26]) we have been able to compute the response time of each task present in the system [17]: this information allowed us to statically analyze the run-time behaviour of the system even if its source code used dynamic binding to resolve invocations.

Those experiments arguably prove that the application of reuse strategies based on inheritance and overriding (and thus dynamic binding) doesn't necessary prevent the system from being statically analyzable for its timing behaviour: this result is attainable in our context thanks to the constraints enforced by the RCM metamodel. The model-based approach to static analysis we chose proved superior to source-based analysis (possibly coupled with code post-processing, as discussed in section 1.1) because it allows a much more precise analysis: the paths we consider are indeed only those that are actually traversed during execution, and not all possible combinations of executions paths.

The acceptance of dynamic binding as a viable mechanism for high-integrity systems obviously opens the door to the use of inheritance and polymorphism too. In particular, (multiple) interface inheritance has enabled us to enforce fine-grained visibility control over the methods provided by the same class: a class can be accessed through one of its implemented interfaces, which provides a partial view of the callable services. Having fine-grained visibility control enforced through inheritance means that only a subset of the possible interactions between two functional models must be verified (on the condition that downcast is not allowed): we are currently investigating on how to exploit this feature to further decrease verification costs.

4 Conclusions

Domain-specific modeling has been successfully applied in several application domains; nevertheless, the high-integrity domain has so far resisted the adoption of that paradigm, allegedly because of the difficulty of exploiting models as a source for verification. In this paper we have shown how a domain-specific metamodel can constrain the design semantics so to allow model-based static analysis of high integrity systems using the dynamic binding mechanism to resolve invocations: domain-specific modeling can thus be used to warrant the preservation of specific properties in the target systems in addition to increasing the productivity of the development process. The main novelty of our approach is not in exploiting radically new technologies or paradigms for DSM, but rather in deriving a DSM directly from a computational model and in exploiting model-based analysis to promote adaptive reuse of object-oriented high-integrity components. The results we achieved with the partners of the ASSERT project (cf. the Acknowledgments at the bottom of the paper) show that the approach is feasible and affordable and it also allows the design and the static analysis of reusable, adaptable, object-oriented components for systems in the aerospace domain: this result may open a new dimension for reuse strategies in the high-integrity domain.

Unfortunately, the most stringent certification standards and the industrial practice for high-integrity systems do *not* consider models as the main verification source, mostly because they date back to an era where conscious model-based approaches (or even mainstream object-oriented languages) were still to come. Some of these standards (most notably DO-178B) are however under revision and they are expected to favor the use of controlled forms of object orientation: the constraints enforced by the RCM metamodel should thus guarantee that next generation high-integrity systems can be designed using advanced, object-oriented, reuse and adaptation techniques.

Acknowledgments. The research work from which this paper has originated was carried out in the ASSERT project (IST-FP6-2004 004033) partially funded by the European Commission as part of the 6^{th} Framework Programme.

The authors gratefully acknowledge Gerald Garcia and Mathieu Le-Coroller (Thales-Alenia Space Cannes) for their support in developing the experimental prototypes and Alessandro Pasetti and Ondrej Rohlik (Swiss Federal Institute of Technology Zurich) for their support in the design of object-oriented functional frameworks.

References

1. Pasetti, A.: Software Frameworks and Embedded Control Systems. Springer-Verlang (2002)
2. Meyer, B.: Design by Contract. Technical Report TR-EI-12/CO (1986)
3. Hall, A., Chapman, R.: Correctness by Construction: Developing a Commercial Secure System. IEEE Software **19**(1) (2002) 18–25
4. RTCA: Radio Technical Commission for Aeronautics. (rtca.org)
5. Gasperoni, F.: Safety, security, and object-oriented programming. SIGBED Rev. **3**(4) (2006) 15–26
6. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer **39**(2) (2006) 25–31
7. SCADE: Safety Critical Application Development Environment. (http://www.esterel-technologies.com/products/scade-suite/)
8. Matlab: (http://www.mathworks.com/)
9. AADL: Architecture Analysis and Design Language. (http://www.aadl.info)
10. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and memory requirements analysis with aadl. In: SigAda '05: Proceedings of the 2005 annual ACM SIGAda international conference on Ada, New York, NY, USA, ACM Press (2005) 1–10
11. Barnes, J.: High Integrity Software - The SPARK Approach to Safety and Security. Addison-Wesley (2003)
12. Bordin, M., Vardanega, T.: Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In: Reliable Software Technologies - Ada-Europe. (2007)
13. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In: Reliable Software Technologies - Ada Europe. (1998)
14. Arlow, J., Neustadt, I.: UML2 and the Unified Process. Addison-Wesley (2005)
15. Burns, A., Dobbing, B., Vardanega, T.: Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. Technical Report YCS-2003-348, University of York (2003)
16. Vardanega, T., Zamorano, J., de la Puente, J.A.: On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels. Real-Time Systems **29** (2005) 59–89
17. Panunzio, M., Vardanega, T.: A Metamodel-driven Process Featuring Advanced Model-based Timing Analysis. In: Reliable Software Technologies - Ada-Europe. (2007)
18. Bordin, M., Vardanega, T.: Automated Model-Based Generation of Ravenscar-Compliant Source Code. In: Proc. of the 17th Euromicro Conference on Real-Time Systems. (2005)
19. 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)
20. RTSJ: The Real-Time Specification for Java. (http://www.rtsj.org)
21. Cechticky, V., Egli, M., Pasetti, A., Rohlik, O., Vardanega, T.: 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)
22. Eclipse: (The Eclipse Modeling project) http://www.eclipse.org/modeling.
23. OMG: UML2 Metamodel Superstructure. (2005)
24. Burns, A., Wellings, A.J.: HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems. Elsevier (1995)
25. oAW: Open ArchitectureWare. (http://www.openarchitectureware.org/)
26. Holsti, N., Langbacka, T., Saarinen, S.: Using a Worst-Case Execution Time Tool for Real-Time Verification of the DEBIE Software. (In: Data Systems in Aerospace 2000 (DASIA 2000))