A UML2 Profile for Reusable and Verifiable Software Components for Real-Time Applications

V. Cechticky[1], M. Egli[1], A. Pasetti[1], O. Rohlik[1] , T. Vardanega[2]

[1] Institut für Automatik, ETH-Zentrum, Physikstr. 3,
CH-8092 Zürich, Switzerland
{eglimi, pasetti, rohlik}@control.ee.ethz.ch
[2] Dept. of Pure and Applied Mathematics, University of Padua
Via Belzoni 7, 35131 Padova, Italy
tullio.vardanega@math.unipd.it

Abstract. Software frameworks offer sets of reusable and adaptable components embedded within an architecture optimized for a given target domain. This paper introduces an approach to the design of software frameworks for real-time applications. Real-Time applications are characterized by functional and non-functional (e.g. timing) requirements. The proposed approach separates the treatment of these two aspects. For functional issues, it defines an extensible state machine concept to define components that encapsulate functional behaviour and offer adaptation mechanisms to extend this behaviour which warrant preservation of the functional properties that characterize the framework. For timing issues, it defines software structures that are provably endowed with specific timing properties and which encapsulate functional activity in a way that warrants their enforcement. A UML2 profile is defined that formally captures both aspects and allows the proposed strategy to be deployed at design level.

1 Introduction

A *software product family* is a set of applications that can be built from a pool of shared software assets. *Software frameworks* [1] offer a way to organize the shared assets behind a product family. They define an architecture optimized for applications in a certain domain and offer predefined components that support its instantiation. During the instantiation process the framework assets are tailored to suit the specific requirements of the target application. To this end, a software framework defines a number of *adaptation points* where application-specific behaviour can be inserted. Most contemporary software frameworks are *object-oriented* in the sense that their reusable assets consist of encapsulated software components and their adaptation mechanisms are based on class extension and interface implementation.

Although software frameworks have proven very successful at fostering a reuse-driven approach in business and desktop applications, they have so far failed to penetrate the realm of hard real-time (HRT) applications. HRT applications are characterized by non-functional (timing) requirements that impose severe constraints on the timing behaviour of the application and that often are mission-critical.

Presently the prevalent paradigm in the real-time world is bases on *model driven architectures*. With this approach, the requirements of the target applications are expressed in a formalism that allows an implementation to be automatically generated from the specification.

Both the reuse-driven and model-driven approaches have strengths and weaknesses. The model-driven approach holds the promise of completely automating the software development process. Additionally, the formal definition of the requirements facilitates formal verification of correctness, for example using model-checking techniques [2]. On the downside, the model-driven approach is intrinsically limited by the expressive power of the modeling language of choice. Model-driven tools also are very costly to develop and their development is only justified for applications that have sizable markets. The reuse approach can be more flexible both because reusable building blocks can, in principle, be provided to cover as wide a range of functionalities as desired, and because it can be applied in an incremental way with repositories of reusable building blocks built up over time. The main drawback with this approach is that the adaptation process is difficult to formalize and developers of critical applications are reluctant to adopt components that they did not develop (the well-known "not-invented-here syndrome") and over whose characteristics they may have little visibility. Furthermore, adaptation techniques are in effect geared to functional requirements only and adaptation to real-time requirements remains poorly understood.

In this paper, we propose a design approach for HRT applications that combines reuse- and model-driven flavours. The proposed approach is reuse-driven in the sense that it sees an application as an instance of an object-oriented software framework. It is model-driven in the sense that a modeling language is defined to describe both the framework components (in terms of their interfaces as well as their behaviour) and their adaptation mechanism. The component implementation is automatically generated from their models. Our approach also allows for the definition of *formally verifiable properties* upon the framework. Since our focus is on real-time applications, we cover both functional and timing properties. Functional properties formalize logical relationships on the variables that define the state of an application. This logical relationship may also be sequential in that it may relate past and present values of the state variables. Timing properties define constraints on the arrival time of external events and on the completion times of application activities.

We regard an object-oriented software framework as a set of interacting components that can be adapted through class extension[1]. The components encapsulate the commonalities of the applications within the framework domain. Their adaptation allows application-specific behaviour to be added to the default behaviour defined at framework level. Figure 1 shows our proposed development process for a framework. We break the process up in three main phases. The *domain analysis phase* defines the target domain of the framework and the functionalities it must provide [3]. This phase is not discussed further in this paper. In the *domain design phase*, the framework components are designed. The output of this phase is a model of the framework. Two

1 The second adaptation mechanism of object-oriented frameworks – adaptation through interface implementation – is seen as a special case of the first as an interface can be represented by an "empty" class (a class with only abstract methods)

views of this model are constructed. The *functional view* defines the framework from a functional point of view. It consists of class diagrams that define the functional architecture of the framework (the component interfaces and their mutual relations) and state charts that define the internal *behaviour* of each component. The functional model also identifies the extension points of the components. The *timing view* defines the HRT characteristics of the framework by identifying and characterizing the threads and the synchronization points and data structures that may be used by applications instantiated from the framework. Finally, in the *domain implementation phase* the components are implemented. For the most part this latter stage is attained by automatic code generation from the component models.

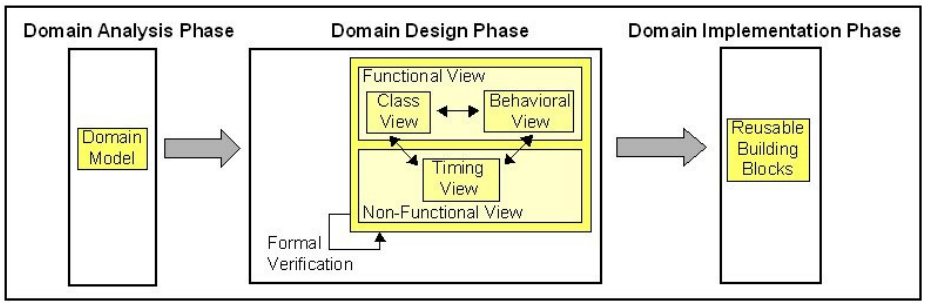

Figure 1: Framework Development Process.

In this paper we discuss the functional and timing views. Notably, these two views do *not* form two separate models. Rather, they provide two different representations of the *one and the same* underlying model. The value of our approach is that it allows these two views to be defined independently of each other, so that conceptually they can be treated in isolation. The concerns arising from each view are merged during the code generation process after verifying the feasibility of the timing view and the correctness of the functional view of the system. The framework model is then processed and the code for the framework components is automatically generated for both the functional and timing dimensions.

The association of verifiable properties to models is typical of model-driven architectures. In a framework context two levels of properties must be distinguished (cf. figure 2). Framework-level properties formalize the commonality of behaviour of applications within the framework domain. These properties must be satisfied by all applications instantiated from the framework. Additionally, each individual application may be endowed with application-specific properties. The adaptation process through which the framework components are tailored to the needs of a target application is constrained to guarantee that the application-level components still satisfy the framework-level properties. The framework instantiation process can thus result in new properties being added but will *never* result in the violation of the framework-level properties.

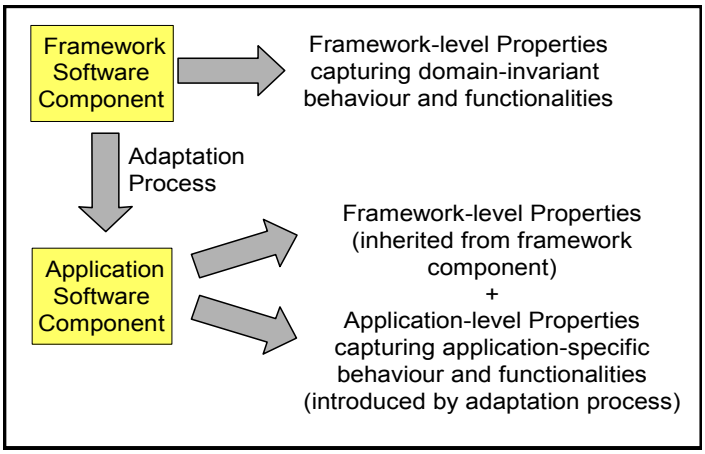

Figure 2: Framework- and Application-Level Properties

In keeping with the standard approach in the model-driven community, the framework modeling language is expressed as a UML2 profile that we name the *FW (framework) Profile*. The bonus of this choice is that UML2 environments are now available that can be customized to enforce a user-defined profile during the design process. Most application developers are familiar with UML-based design. The transition to our approach can therefore take place naturally and at minimal costs.

The remainder of this paper is organized as follows. Sections 2 and 3 describe the design approach for, respectively, the functional and timing aspects of a framework. Section 4 presents the code generation approach. Section 5 describes a case study, discusses related work, and concludes the paper.

The full definition of our framework profile and associated development process. can be found in [4]. A plug-in for the Eclipse UML2 platform to enforce the profile during the design process is available from [5].

2 Functional Design

The functional design of the framework consists in the definition of its functional view. The functional view describes the framework architecture, the functional behaviour of the framework components, and the component adaptation mechanisms. The framework architecture is described in terms of the external interfaces of the framework components and their mutual interconnections. The framework architecture is represented through UML2 class diagrams. The adaptation mechanism is also defined on the class diagrams by identifying the class methods that are either abstract or virtual. The functional behaviour of the components is represented through UML2 state charts. Advanced support of state charts is the main reason of choice of UML2 over older versions of UML.

The FW Profile defines the rules that constrain the way UML2 class diagrams and state charts are built. The description of the FW Profile is best given in terms of three elements: (i) the restriction of the UML2 state machine model; (ii) the component

extension mechanism; (iii) the action language to define the actions associated to the state machines.

2.1 UML2 State Machine Restrictions

The rationale for restricting the UML2 state machine model stems from our intent to use state machines solely to describe the functional part of the behaviour of a class. Behaviour that is time-related (e.g. waiting for an event) or that implies interaction across thread boundaries (e.g. engaging a synchronization with a thread of control in another class) is modelled in the timing view using other mechanisms discussed in section 3. Accordingly, the FW Profiles stipulates that state transitions can only be triggered by calls on the operations defined by the class associated to the state machine. Other types of transition mechanisms (through signals or time triggers) are forbidden. By the same token, no events are allowed by the FW Profile.

In practice, the state machines are only used to model behaviour *inside* threads. Removing the time dimension from the state machine models is important because it removes all the semantic ambiguities that plague the UML2 state machine model and that make other attempts to use state machines to model behaviour unwieldy [15, 16].

The second driver for the restrictions on the UML2 state machine model is simplicity and elimination of unnecessary features. This helps us streamline the model validation and the code generation processes. UML2 allows for three kinds of states: simple states, composite states, and submachine states. Only simple and composite states are allowed by the FW Profile. UML2 also defines several kinds of pseudo-states but the FW Profile only retains the initial pseudo-state and the choice pseudo-states. Finally, UML2 allows *entry*, *do* and *exit* actions to be associated to states. The *entry* and *exit* actions are retained but the *do* action is not necessary since the FW Profile state machines are purely reactive: they only do something when they are triggered by a call to a trigger operation defined on the class to which they are associated.

2.2 Component Extension Mechanism

At class level, component extension is modelled through class extension. The main constraint on the extension mechanism is that it must allow new properties to be defined on the extended component while preserving the properties defined on the base component (see figure 2).

Figure 3 illustrates the proposed mechanism. Class `Base` represents a component provided by the framework. Class `Derived` represents the adapted component constructed during the framework instantiation process. The framework-level properties capture aspects of the `Base` state machine topology and of its state transition logic. The FW Profile ensures that these properties are preserved by constraining the extension to define the internal behaviour of one or more of the states of the base state machine without altering its topology and transition logic. This is illustrated in the figure where the derived state machine differs from the base state machine only in including an embedded state machine that adds to the base states. The derived state machine defines the internal behaviour of a state that was initially defined as being a simple state.

The FW Profile adopts the extension process of figure 3 and forbids all other kinds of

state machine extensions that are allowed by UML2 (redefinition of transition, definition of new transitions between existing states, definition of new states, etc).

In order to freeze the transition logic of a state machine, the FW Profile stipulates that the trigger operations that control the state transitions must be defined as final (i.e. they cannot be altered during the class extension process).

In order to ensure the preservation of properties defined on the base state machine, the two state machines – the base state machine and the state machine embedded in one of its states during the extension process – must be decoupled: trigger operations defined on the derived class must act on one and only one of the two state machines, and the embedded state machine must not be allowed to trigger transitions in the base state machine.

The extension mechanism enforced by the FW Profile, though very simple, corresponds to a realistic situation that often arises in framework design. This is the case described by the well-known template design pattern where a class defines some skeleton behaviour that offers hooks where application-specific behaviour can be added by providing implementation for abstract methods. The behaviour encapsulated in the skeleton, however, is intended to be invariant. In terms of the FW state machine model, the invariant skeleton behaviour is encapsulated by the base state machine whereas the variable hook behaviour is encapsulated by the nested state machines added by the derived class.

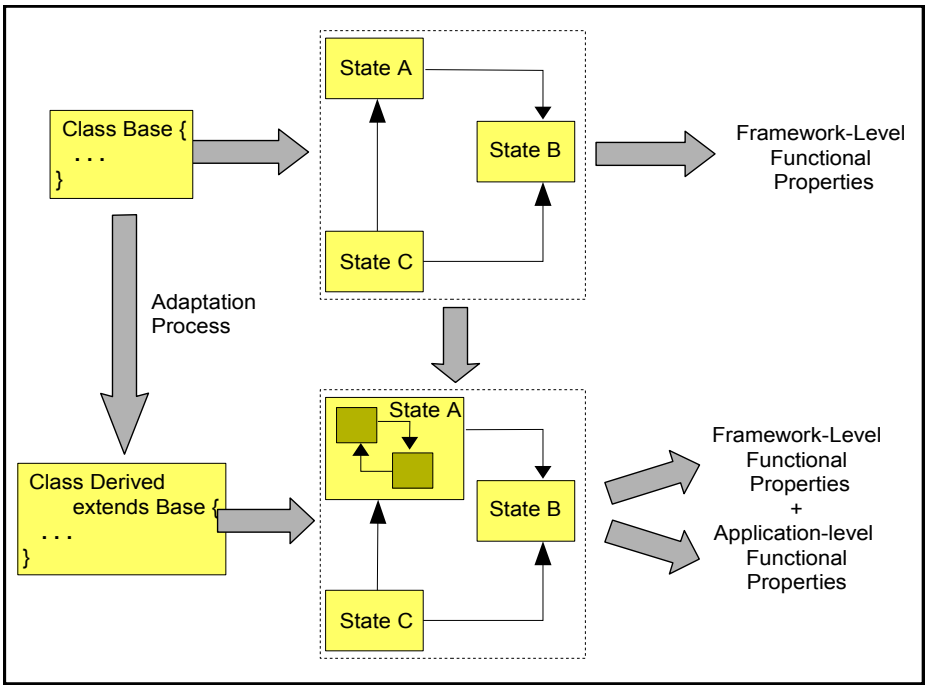

Figure 3: Framework Component Extension Mechanism

2.3 Action Language

The FW Profile stipulates that state machines are used to model the behaviour of a class. At its most basic level, the link between a class and its associated state machine is defined by the trigger operations (the class operations that trigger transitions in the state machine). In order to allow for a more complete link between a state machine and its associated class, the profile also defines an action language. The FW action language is introduced to define: transition guards, transition effects, and state entry and exit actions. It allows manipulation of class methods, class attributes of integer or boolean type, and references to associated class instances. It thus allows a guard to be expressed as a Boolean expression combining the above elements and transition effects.

3 Timing Design

A software framework is not an executable application and hence it cannot in general be subject to timing requirements. By arguing that a framework supports the instantiation of HRT applications, we maintain that applications instantiated from it are "aware" of the timing requirements imposed upon them and that their ultimate feasibility can be statically analyzed against those requirements. The latter property is of course crucial to the mission critical domain of our interest. Our goal is thus to offer a design approach that guarantees that all applications instantiated from a certain framework are statically analyzable for their timing properties.

The approach we take to this effect is centered on a reuse-geared adaptation of the HRT-UML design method [7,8,9]. A distinct prerogative of HRT-UML is that it adopts a concurrent computational model based on the Ravenscar Profile [10]. The Ravenscar Profile amounts to a set of restrictions placed on the concurrent behaviour of a system. Notably, such restrictions are genuinely orthogonal to the FW Profile presented in this paper since they do *not* concern the functionality that can be expressed by the sequential part of target programming languages, but only the concurrent behaviour of the application. All concurrent applications designed in compliance with the restrictions of the Ravenscar Profile are statically analyzable for their timing behaviour at run time by construction.

If the Ada programming language [6] is used, Ravenscar compliance can be proven *a posteriori* on the source code submitted to the compiler. This assurance is an important asset for model-based code generators, since code accepted by a Ravenscar-aware compiler is guaranteed to behave at run time exactly as assumed by static analysis. Yet HRT-UML adds considerable value to this assurance by elevating the Ravenscar restrictions to the design level, so that the model space itself warrants structural compliance with them by construction, that is *a priori*.

HRT-UML does so by placing rigorous restrictions (which all emanate from the Ravenscar Profile) on the ontology and the taxonomy of the allowable elements of a model. Ontological restrictions specify the semantic nature of model elements (hence what they are for and how they can be used). Taxonomical restrictions define the allowable relations that can be placed among model elements (hence how they can interconnect to one another). It is out of the scope of this paper to provide an

exhaustive presentation of HRT-UML. In the following we will simply illustrate the basic principles of HRT-UML design that are relevant for illustration of the FW Profile presented in this paper. For further details on HRT-UML the reader is referred to the relevant literature [7,8,9,10].

3.1 Brief Ontology of HRT-UML Model Elements

Each element of an HRT-UML model is a cohesive aggregate of:

- one **Provided Interface** (PI), which publishes: (i) the signature of the services (operations) that the element is capable of executing on request from the outside environment; (ii) the constraints placed on invocation of its operations (the *invocation protocol*); and (iii) the time bound (WCET) stipulated on the execution of the required operation
- one **Object Control Structure** (OBCS), which is the agent responsible for execution of the invocation protocol attached to the invocation of PI operations
- one **Thread** that associates an autonomous run-time behaviour to the element; such Thread is associated to a thread of control whose run-time behaviour amounts to a non-terminating iteration revolving around a *single* activation event arising from either a hardware interrupt (a clock or some other device with a stipulated behaviour) or software
- one **Operation Control Structure** (OPCS) per operation published in the PI, which provides the functional specification of the operation; as a direct consequence of the Ravenscar restrictions, such functional specifications must involve *no* internal concurrent action and *no* voluntary suspension
- one **Required Interface** (RI), which publishes to the outside environment: (i) the signature of the operations that the element needs to use for carrying out its own duties; (ii) the execution protocol that the element is willing to accept for their invocation; and (iii) the execution time bound (WCET) that warrant the preservation of the corresponding bounds on the PI.

Not all model elements need to possess all of the above internals. Specific ontological rules determine the internal composition of each model element and the nature of each internal constituent. Intrinsic to the hierarchical nature of HRT-UML design, the primary ontological distinction is to be made between non-terminal and terminal elements.

Non-terminal elements are “capsules” that hide their inner detail to the outside and only present PI and RI (both of which can be void). Terminal elements are fully resolved and allow/require no further decomposition. A non-terminal entity is created by either top-down decomposition of a parent element into a set of child elements or by bottom-up aggregation of independent sibling elements into a containing, hierarchically superior, element. Hierarchical decomposition requires that the PI of the parent element be *delegated* to matching PI of child elements. Hierarchical aggregation permits to *promote* selected items of the PI of the aggregated elements to the PI of the aggregating element and hide all others. Decomposition and aggregation must preserve respectively respect the ontology of model elements. In other words,

specific semantic rules determine the legal decompositions and the allowable aggregations.

A further ontological rule on the composition of model entities stipulates that only "active" terminal elements include a Thread. In fact, terminal elements that include a Thread are denoted Cyclic or Sporadic. The former implies that the thread of control associated at run time to the Thread entity of the Cyclic element takes its activation event from a fixed-rate clock event. The desired rate can be changed at run time by the application logic (as it would occur during a "mode change" situation) as long as the PI of the corresponding Cyclic element publishes an operation to that effect. The latter implies that the source of the activation event be other than time, however requiring a guaranteed minimum separation between two subsequent arrivals of it. This property is assumed at model level and warranted by the model transformation rules that inform automated code generation.

A few words are in order on the ontology of the internal constituents of HRT-UML model elements. The PI of a terminal element is delegated to the PI of the element's OBCS, which may hold further operations to match any of those appearing in the RI of the element's Thread, if any. While having an empty PI, the Thread's RI includes the operations to fetch service requests from the OBCS as well as the invocation of those services in the PI of the corresponding OPCS. The element's OPCS, finally, has both PI and RI, the latter because the execution of a service charged to the element may need to use the services provided by other visible elements of the system. Figure 5 illustrates these notions by depicting the ontology of a Cyclic element.

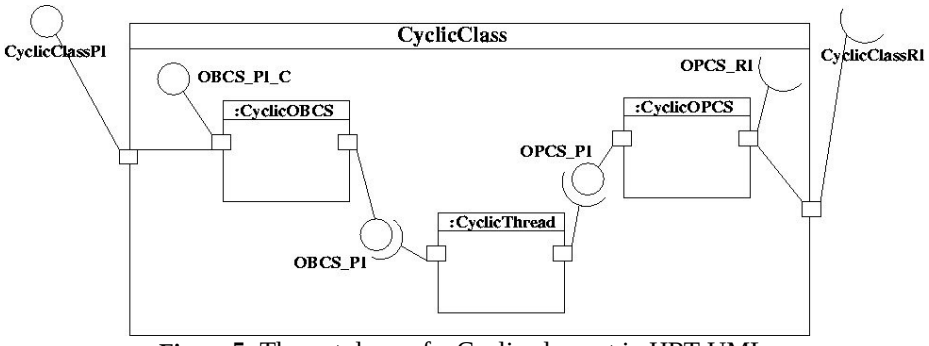

Figure 5: The ontology of a Cyclic element in HRT-UML.

3.2 Brief Taxonomy of HRT-UML Model Elements

The HRT-UML invocation protocols address two dimensions of critical relevance to concurrent computing: (i) the guarantee of mutual exclusion on access to element internals on execution of a given PI operation; and (ii) the suspension of the call until the element internal state permits to execute it. The former caters for controlled access to shared resources. The latter corresponds to placing a state-dependent conditions akin to Dijkstra guards on the servicing of PI operation [12] and thus adds power to the simple form of protection warranted by the former.

HRT-UML draws from the Ravenscar Profile the restrictions on the use of these invocation constraints. Firstly, it assumes that the locking policy in force in the execution environment be based on Immediate Priority Ceiling Inheritance [11,12]. Secondly, it prescribes that no more than *one* state-dependent invocation constraint should ever appear in the PI of a terminal element so as to avoid the non-determinism that would incur from multiple guard conditions becoming open simultaneously on one and the same PI. Thirdly, it requires that no more than one call should ever queue at any one time awaiting access to a state-constrained operation. The intent of this very severe restriction is to *constructively* avoid the non-determinism that would arise from call queuing. HRT-UML takes a very straightforward approach to enforcing this particular restriction: it stipulates that state-constrained operations should only be invoked by the Thread of Sporadic elements. In essence, PI invocation protocol of a Sporadic element would specify how a given sequence of PI invocations would produce the activation event of the element's Thread.

3.3 Integration with the FW Profile

HRT-UML defines the concurrent architecture of the framework. That architecture may include components defined to the level of terminal elements, as either fully developed object instances or plain classes. Other components may be left to the stage of non-terminal elements, hence simply described by the corresponding PI and RI (either of which may be empty).

The FW Profile defines the functional components of the framework architecture. As discussed above, HRT-UML stipulates that such functional specifications pertain to the PI, along with the applicable invocation protocol, to the RI, and to the OPCS components of model elements. In practice, the FW Profile imports the HRT-UML constraints on those ontological components and guarantees that they are respected throughout design and instantiation. On these conditions, verification on the model can safely be performed for the timing and the functional aspects in isolation, while code generation can also be undertaken separately for the structural (concurrent) and the functional part.

4 Code Generation

The core of the approach proposed in this paper is a UML2 profile that allows functional and timing behaviour of extensible components to be expressed in a manner that permits the inclusion of components in HRT applications. Automatic generation of the component code from its profile-compliant models is a natural extension to this objective.

We do not deem it appropriate to define a *single* code generator for our profile. Different domains have different coding rules and must interface to different middleware or operating systems. The code generator must therefore be framework-specific. The *code generating approach* can, however, be generic.

The approach we propose propagates the split between functional and non-functional issues down to code level. We have found it convenient to have two code generators (cf. Figure 6). The first one processes the timing view of the framework model and generates a set of *structural containers* that enforce the timing constraints. In practice,

these containers implement HRT-UML entities (sporadic, cyclical or protected elements) with no functional behaviour attached to them. The second code generator processes the functional view of the framework model and generates the classes that implement the state machine logic, which encapsulates the functional behaviour of the framework components. Each state in a state machine is mapped to an instance of a generic `State` class. There is an aggregation relationship between a class and the states of the associated state machine. Any state can embed further state machines and thus embedded states (cf. figure 7). Triggers are mapped to parameterless operations that operate on a state by calling operations `unmarkAsCurrent()` and `markAsCurrent()` on `State` objects. Guards, transition actions and state entry and exit actions are expressed in the model using the action language associated to the FW Profile. The action language syntax is simple and compatible with most mainstream object-oriented languages.

The two code generators are integrated in the sense that the functional code is designed to be embedded within the structural containers. One can imagine that the non-functional code generator generates a set of non-functional containers, whereas the functional code generator fills the containers with the functional code.

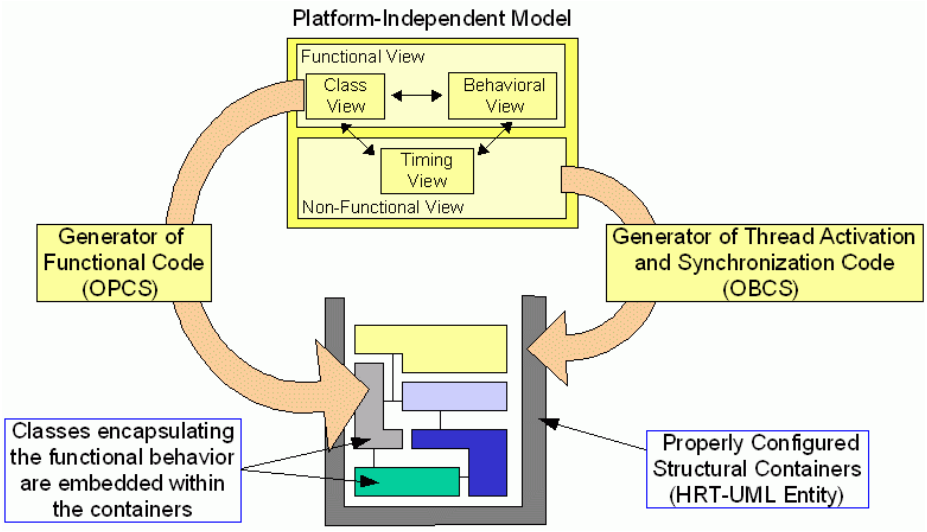

Figure 6: Code Generation Approach

The Eclipse Java Emitter Templates (JET) is the basic technology for our code generator. JET is an open source tool for code generation. It is a generic template engine that permits to generate any type of source code. A model-to-code generator for Java has been implemented and the development of a generator for Ada 2005 [6] is now under way, which builds on the work described in [13,14]. Both generators are distributed as plug-ins for Eclipse and for IBM RSM [5].

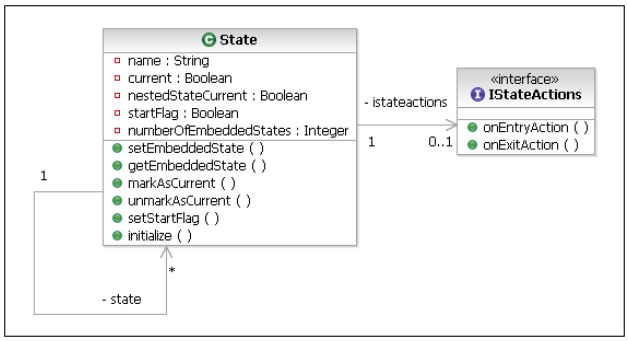

Figure 7: Hierarchical State Machine Implementation

5 Case Study and Conclusions

We are using our approach to construct a software framework for satellite on-board applications. These applications form a good case study since they are subject to HRT constraints and are mission-critical. Our framework covers the handling of *telecommands*, namely the commands that are sent to a satellite by the ground station.

Telecommands are characterized by timing requirements that define the minimum inter-arrival time of consecutive telecommands and the maximum execution time of each telecommand. At a functional level, telecommands are characterized by some requirements that apply to all satellite applications, and by others that are specific to each satellite application. Examples of the former are the requirements that telecommands must report the outcome of their execution to the ground station; that they must perform an acceptance check that may lead to their being rejected before their execution starts; or that their execution can be aborted by the ground station; etc.

In accordance with the process proposed in this paper, we designed the framework in two steps. In one step, we defined the timing view to implement the timing requirements. This was done by defining a set of HRT-UML sporadic structural containers to hold groups of telecommands with the same timing constraints. This architecture ensures that timing requirements are satisfied, independently of the functional content of the telecommands. In the other step, we defined the functional view of the framework by defining a set of reusable components. Their behaviour was described by state machines. The state machine logic ensures that the application-invariant functional requirements of the telecommands are satisfied. Since the design complies with the FW Profile, the application developer can extend these components to implement the application-specific behaviour in the knowledge that properties such as reporting of execution outcome, or implementation of an acceptance check will be preserved.

The case study demonstrated the four decisive advantages of our approach over rival approaches to framework-based software reuse. Firstly, the functional and non-functional aspects of the framework are defined separately from each other and are only merged when the models are translated into code. This simplifies the design

process. In fact, functional and non-functional design can be entrusted to two different teams (as was in fact done in our case study). Unlike other authors who have attempted to use state charts to model both the functional and non-functional behaviour of real-time applications [15,16], we use two different modelling vehicles for functional and non-functional issues and thus avoid the semantic uncertainties and complexities of the UML2 state machine concept.

Secondly, the encapsulation of our approach in a UML2 profile means that compliance with the approach can be enforced at design time using standard software design tools. It also means that translation to code can be easily automated since there are standard ways of building code generators for UML2-based models.

Thirdly, compliance with our FW Profile ensures that functional and timing properties defined at framework level are preserved when the framework components are reused to instantiate a particular application. This is an essential pre-requisite for software reuse in mission-critical applications and we are not aware of other methodologies that provide the same guarantee.

Finally, the formulation of functional properties on profile-compliant UML2 models opens the way to their formal verification at model level. This is an avenue that we are currently exploring in a follow-on project. The final objective is to arrive at a reuse methodology where reusable components are provably endowed with functional properties whose preservation be provably guaranteed throughout the adaptation process.

Acknowledgments

The definition of HRT-UML design method is the result of the collective effort of several people. The authors of this paper gratefully acknowledge the considerable contributions by Daniela Cancila and Enrico Mezzetti from the University of Padua (Italy), and Silvia Mazzini, Stefano Puri and Maria Rosa Barone from Intecs (Italy).

References

1. Pasetti, A.: Software Frameworks and Embedded Control Systems, LNCS Vol. 2231, Springer-Verlag, 2002
2. Wang F.: Formal verification of timed systems: A survey and perspective. Proceedings of the IEEE, 92(8), pages 1283-1305. August 2004.
3. Cechticky, V., Pasetti, A., Rohlik, O., Schaufelberger, W.: XML-Based Feature Modelling, in Bosch, J., Krueger, C. (eds), Software Reuse: Methods, Techniques, and Tools (ICSR), LNCS Vol. 3107, Springer-Verlag, 2004.
4. Cechticky, V., Pasetti, A., Rohlik, O., Vardanega, T: Automated proof-based System and Software Engineering for Real-Time Applications: Framework Design Report. Technical Report, 2005. Available at ASSERT project website: http://www.assert-online.org/
5. Cechticky, V., Pasetti, A., Rohlik, O.: The Model-to-Code Transformation Project website http://people.ee.ethz.ch/~ceg/assert/model2code/
6. ISO SC22/WG9: Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1 (Draft 16). (2006) Available at http://www.adaic.com/standards/rm-amend/html/RM-TTL.html.

7. Mazzini S., D'Alessandro M., Di Natale M., Lipari G., Vardanega T.: Issues in Mapping HRT-HOOD to UML. In: Proc. 15th Euromicro Conference on Real-Time Systems, IEEE, 221-228, July 2003 (ISBN: 0-7695-1936-9).
8. Mazzini S., D'Alessandro M., Di Natale M., Domenici A., Lipari G., Vardanega T.: HRT-UML: Taking HRT-HOOD onto UML, Reliable Software Technologies Ada Europe 2003, Springer Verlag. LNCS(2655): 405-416, June 2003 (ISBN: 3-540-40376-0).
9. Vardanega T., Di Natale M., Mazzini S., D'Alessandro M.: Component-Based Real-Time Design: Mapping HRT-HOOD to UML, IEEE CS Press, In: Proc. 30th Euromicro Conference, pp. 6-13, September 2004 (ISSN: 1089-6503).
10. Vardanega T., Zamorano J., de la Puente J.A.: On the Dynamic Semantics and the Timing Behaviour of Ravenscar Kernels, Real-Time Systems, 29(1):5989, 2005. Kluwer Academic Publishers (ISSN: 0922-6443).
11. Goodenough, J., and Sha, L. The priority ceiling protocol: a method for minimizing the blocking of high priority Ada Tasks. Technical Report SEI-SSR-4, Software Engineering Institute, Pittsburgh, Pennsylvania, 1988.
12. Dijkstra, E. 1975. Guarded commands, nondeterminacy and formal derivation of programs. CACM 18(8): 453– 457.
13. M. Bordin, T. Vardanega: Automated Model-based Generation of Ravenscar-compliant Source Code, In: Proc. 17th Euromicro Conference on Real-Time Systems, July 2005, IEEE, 69–77 (ISBN: 0-7695-2400-1, ISSN:1068-3070).
14. Bordin M., Vardanega T.: A New Strategy for the HRT-HOOD to Ada Mapping, Reliable Software Technologies – Ada-Europe 2005, Springer. LNCS(3555):51–66, June 2005 (ISBN: 3-540-26286-5).
15. Ober I., Graf S., Ober I.: Validating timed UML models by simulation and verification. In STTT, Int. Journal on Software Tools for Technology Transfer, Springer 2005.
16. Latella D., Majzik I., Massink M.. Automatic verification of a behavioral subset of UML statechart diagrams using the SPiN model-checker. Formal Aspects of Computing, (11), 1999.
17. Packet Utilization Standard, European Space Agency, ESA PSS-07-101, (ECSS version ECSS-E-70-41). Available from: http://www.ecss.nl/forums/ecss/_templates/default.htm?target=http://www.ecss.nl/forums/ecss/dispatch.cgi/standards/showFolder/100004/def/def/a492