A UML Profile for Designing Reusable and Verifiable Software Components for On-Board Applications

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

[1] Institut für Automatik, ETH-Zentrum, Physikstr. 3,
CH-8092 Zürich, Switzerland
tel: +41 44 632 7812, fax: +41 44 632 1211
{eglimi,pasetti,rohlik}@control.ee.ethz.ch
[2] Dept. of Pure and Applied Mathematics, University of Padua
Via Belzoni 7, 35131 Padova, Italy
tel: +39 049 827 5859, fax: +39 049 827 5892
tullio.vardanega@math.unipd.it

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 on-board applications taken in the ASSERT project. On-board applications are characterized by functional and non-functional (timing, in particular) 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 while warranting the preservation of the functional properties defined at the level of the framework. For timing issues, it defines software structures which are provably endowed with specific timing properties and which encapsulate functional activity in a way that enforces them. A UML2 profile is defined to formally capture both the state machine concept and the real-time structures. The profile allows the proposed approach to be enforced at design level. We are using our approach to construct a software framework for satellite on-board applications.[*]

A *software product family* is a set of applications that can be built from a pool of shared software assets. *Software frameworks* 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 pre-defined components that support its instantiation. During the instantiation process, the assets provided by the framework are tailored to match the specific requirements of the target application. For this purpose, 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 other desktop applications, they have so far failed to penetrate the world 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 (at least) mission-critical.

At present, the prevalent paradigm in the real-time world is based 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 application 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 opens the way to formal verification of their correctness using, typically, model-checking techniques. On the downside, the model-driven approach is limited by the expressive power of the selected modeling language. Model-driven tools, moreover, are extremely costly to develop and their development is justified only for applications that have large 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 being built over time. The main drawback of this approach is that the adaptation process is difficult to formalize and developers of critical applications are reluctant to adopt components which they did not develop and over whose characteristics they have little visibility. Also, adaptation techniques are geared to functional requirements and adaptation with respect to real-time requirements remains poorly understood.

In this paper, we propose a design approach for HRT applications that *combines reuse- and model-driven* elements. 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 and their adaptation mechanism. This modeling language describes the framework components in terms of their interfaces, their behaviour, and their adaptation mechanisms. The component implementation is automatically generated from their models. Our modeling approach also allows the definition of formally verifiable *properties* upon the framework. Since our focus is on real-time applications, both functional and timing properties are covered. Functional properties formalize logical relationships on the variables that define the state of an application. This logical relationship may also be sequential in the sense 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.
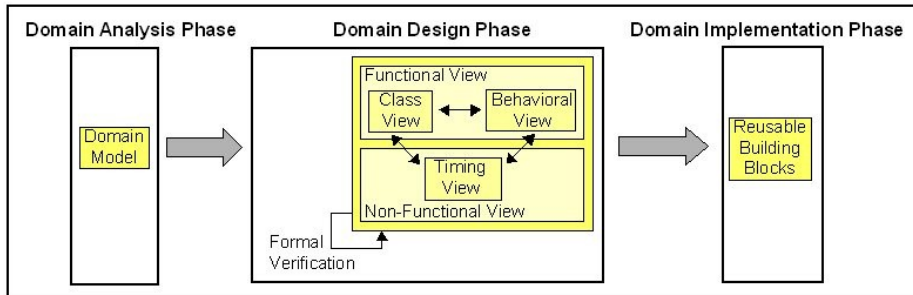

Figure 1: Framework Development Process

We see an object-oriented software framework as a set of interacting components that can be adapted through class extension. 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). 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. Three phases are recognized. In the *domain analysis phase*, the target domain of the framework and the functionalities it must provide are defined. This phase is not considered

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 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 interrelations) 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. This is done by identifying and characterizing the threads and the synchronized data structures that may be used by applications instantiated from the framework. Finally, in the *domain implementation phase* the components are implemented. This is mostly done by automatic code generation from the component models.

This paper focuses on the definition of the functional and timing views. It is important to stress that these views do not form two distinct models. Rather, they provide two different representations of the *same* underlying model. The value of our approach is that it allows these two views to be defined independently of each other. In this sense, functional and timing aspects can be said to be treated separately. The concerns expressed by the functional and timing views are merged during the code generation process (upon verifying the feasibility of the timing view and the correctness of the functional view of the system) when the framework model is processed and the code for the framework components is automatically generated to implement both the function

The association of verifiable properties to models is typical of model-driven architectures but, in a framework context, two levels of properties must be distinguished (see 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 match the needs of a target application is constrained to guarantee that the application-level components still satisfy the framework-level properties. Thus, the framework instantiation process can result in new properties being added but will *never* result in the violation of the framework-level properties.
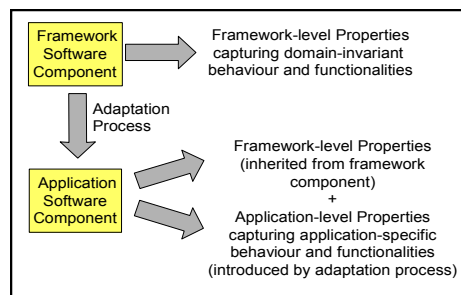

Figure 2: Framework- and Application-Level Properties

In keeping with the standard approach in the model-driven community, the modelling language to describe the framework is expressed as a UML2 profile which we call the *FW 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 adoption of our profile can therefore be more natural and more easily attain compliance with additional design constraints and rules. A plug-in for the IBM Rational Software Modeler and Eclipse UML2 platform to enforce the profile during the design process is also available.

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 represented through UML2 class diagrams. Each component is represented by one or more classes. The functional behaviour of the components is represented through UML2 state charts. A state chart is associated to each class.

The FW Profile defines the rules that constrain the way in which UML2 class diagrams and state charts are built. The main motivation for restricting the UML2 state machine model is the fact that we use state machines exclusively 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. 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.

At class level, component extension is modelled through class extension. The FW Profile adds a state machine extension mechanism. 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). The proposed mechanism is illustrated in figure 3. 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 adding an embedded state machine to one of the base states.

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).
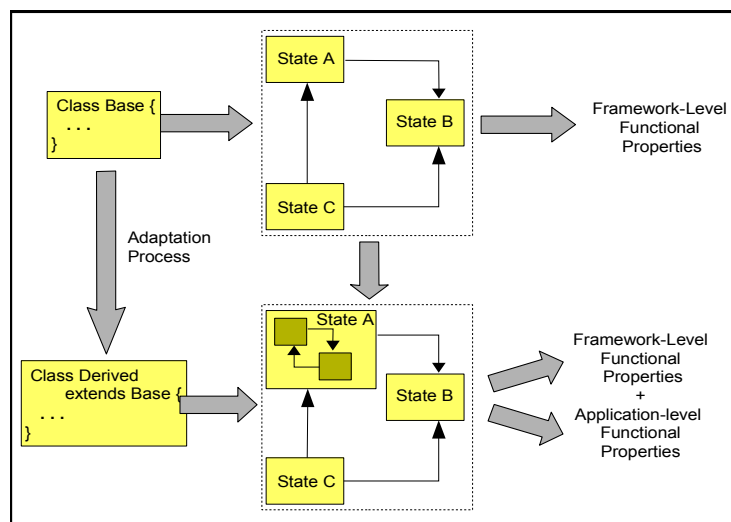
Figure 3: Framework Component Extension Mechanism

A software framework is not an executable application and hence it cannot in general be subject to timing requirements. When we say that a framework supports the instantiation of HRT applications, we mean that the applications instantiated from it are subject to timing requirements and that their feasibility can be statically analyzed against them The latter property is of course crucial to the mission critical domain of our interest. Our goal in this regard 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 end is centred on a reuse-geared adaptation of the HRT-UML design method which adopts a concurrent computational model based on the Ravenscar Profile.

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 makes inclusion of the components in HRT applications possible. Against this background, it is natural to consider automatic generation of the component code from its profile-compliant models.

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 that it is convenient to have two code generators. The first one processes the timing view of the framework model and generates the container structures that enforce the timing constraints. The second one processes the functional view of the framework model and generates the classes that implement the state machine logic that encapsulate the functional behaviour of the framework components. The two generators are integrated in the sense that the functional code is designed to be embedded within the containers.