Issues in Mapping HRT-HOOD to UML

Silvia Mazzini, Massimo D'Alessandro
Intecs HRT
via L. Gereschi 32-34
I-56124 Pisa, Italy
silvia.mazzini@pisa.intecs.it

Marco Di Natale, Giuseppe Lipari
Scuola Superiore S.Anna
viale R. Piaggio 34
I-56025 Pontedera, Italy
{marco,lipari}@sssup.it

Tullio Vardanega
Università di Padova
via G. Belzoni 7
I-35131 Padova, Italy
tullio.vardanega@math.unipd.it

Abstract

HRT-HOOD has methodological strengths that deserve to be preserved in the face of the commercial decline of HOOD technology. The UML meta-model, on the other hand, has a level of flexibility that makes it an especially attractive platform to express the specific real-time design minded features of the HRT-HOOD method. The object-oriented connotation of the method that results from mapping HRT-HOOD onto UML raises methodological issues that we deem of interest to the real-time community at large. This paper discusses three such issues in particular: the prevalence of objects over classes in real-time design, with the consequent inversion of the standard object-oriented development paradigm; the need to derive classes "by example", which arises from the demand to allow multiple, yet static, instances of real-time objects initially designed as singleton; the opportunity of reuse-oriented component-based real-time development, which descends from using interfaces instead of classes as the target of associations among objects.

1. Introduction

A query for HRT-HOOD [3, 4] on popular search engines currently scores in the range of 300 to 600 significant hits. This score arguably attests the varied use that HRT-HOOD has attained as a reference term, a teaching aid and an industrially-applied method.

Albeit originated from within the niche market of the somewhat esoteric HOOD community, HRT-HOOD exhibits two distinguishing features that make it especially significant to the real-time domain: (**i**) it encompasses a well-defined process model, with the discipline and rigour that are paramount to the development of high-integrity real-time systems; (**ii**) it promotes design conformance to a computational model that facilitate static timing analysis; a most attractive instance of computation model especially suited for HRT-HOOD is the recently emerged Ada Ravenscar Profile [1, 2], the consolidation process of which was effectively prompted by the definition of the method.

The simultaneous occurrence of the commercial obsolescence of the HOOD technology offer (not the discipline, though!) and the massive advent of the UML one [10] offers the opportunity for a thorough reflection on whether and how the "goodies" of HRT-HOOD can be migrated to and, possibly, augmented by the UML meta-model platform.

Exploring this opportunity was the objective of a research project recently sponsored by the Italian Space Agency, with the authors as the main actors. This project defined an HRT-UML method that preserves the distinguishing features of HRT-HOOD while also importing useful characteristics from pure object-oriented design.

We believe that the embedding of selected object orientation characteristics into the process of real-time design raises issues of interest to the real-time community at large. In this paper, we undertake to illustrate the nature of those issues and the way we have addressed them in the definition of the HRT-UML method.

2. Setting the Scene

The UML has been designed from the outset as a wide-ranging, general-purpose modeling language, aimed to a variety of application domains, including business and other non-software processes.

Inclination towards generality was by no means unique to UML. Since their inception in the late 80s, in fact, most object-oriented methods have assumed generality and flexibility as paramount, with a view to addressing the needs of the widest possible audience.

Domain-specific communities, though, among which the real-time systems one, have constantly countered this pull with the demand for more specific (i.e. more semantically characterised) concepts and notations.

Several methods have attempted to address the specific needs of real-time systems analysis and design.

The HRT-HOOD method is one such method. Its definition responded to the customer requirement (the European Space Agency) to base upon the HOOD method [7] at the then-current version 3.1. A subsequent effort by the HOOD user group progressed the method definition to a more extensively object-oriented flavored version 4.0 [8]. Revision 4.0 did not attempt to address the real-time concerns, though, simply referring users with that concern to HRT-HOOD. All references to the base HOOD method in this paper shall therefore be understood as referring to HOOD v3.1.

The founding principle of HOOD was hierarchical, rigorously top-down development. HOOD uses the *object* as the main design abstraction. A HOOD object is an encapsulated unit fully defined by its provided and required interfaces, and internally structured as a set of co-operating child objects. The interfaces include operations that can be annotated with information on the associated data flows and possible exceptions, and with the synchronisation semantics of the operation invocations.

A modern component-based methodology does require the formal definition of interface, functionality and structure. Interestingly, HOOD serves this aspect (at least for the required interface and structure definitions) much better than UML does. Extending UML to capture this component-oriented features of HOOD thus became one of the main strands of our effort.

Objects in HOOD may be *active* or *passive*: an active object possesses an own thread of control, and the execution of its provided operations may be *constrained* by the current internal state of the object. HOOD objects have a structured textual description, called ODS for Object Description Skeleton, which effectively defines the meta-model of the method. The ODS may also contain a description of the object behaviour in any formalism of liking to the designer, the typical choice being state machines, for which HOOD offers a graphical notation called Object State Transition Diagram.

HRT-HOOD significantly refined the HOOD object definition, with the intent of tailoring the expressive power of the design activity to the needs of real-time systems. HRT-HOOD achieved this goal by purposefully shedding (and deprecating) generality from the design abstractions. To this end, HRT-HOOD made two distinctive choices: (**i**) firstly, it incorporated explicit design and analysis support for the abstractions that are typically required by real-time system designers; (**ii**) secondly, it constrained the logical architecture design activity (which conveys the functional view of the system) so that it conform, by construction, to a computational model that facilitates timing analysis.

HRT-HOOD broke the concept of active object down into that of *cyclic* and *sporadic*, each with specific operation and synchronisation semantics and restrictions on the allowable provided and required interfaces, and introduced the concept of *protected* object to permit controlled sharing of resources. With this choice, HRT-HOOD effectively captured the most fundamental design abstractions to be encountered in the development of real-time systems.

HRT-HOOD also augmented the HOOD ODS to permit the capture and the expression of the *real-time attributes* of the object (e.g.: period, deadline, priority, worst-case computation time).

The fostered elicitation of the real-time attributes of objects and their semantic conformance to a specific computational model allow HRT-HOOD designs to be analysed for their schedulability since early in the development cycle. A particularly attractive computational model for HRT-HOOD is the Ada Ravenscar Profile, in conjunction with deadline monotonic priority assignment and response time analysis, which is amply discussed in [2].

The close tie to a language-supported computational model represents an outstanding asset of HRT-HOOD over other real-time design methods, for the intended semantics of its design components is faithfully preserved in the transition from design to code.

HOOD (and likewise HRT-HOOD) objects model the main structural components of a system, those that have a control and computational role. Data are modeled by the types of the target implementation language (which Ada 83 [9] was for HOOD) or, recently, by user-defined *classes*. In fact, the notion of class is somewhat ill-fitted in HOOD, which really is object-based as it lacks support for true inheritance and polymorphism among objects. HOOD objects are notionally *singleton*, that is, classes that allow a single instance only. Data types are an orthogonal dimension to a HOOD design, which effectively is left out from architectural modeling and deferred to the detailed design and coding phase, where target language code can be embedded in the ODS. This deficiency has recently been addressed in the 4.0 revision of the HOOD method, which is however of little consequence to HRT-HOOD that is based on the earlier 3.1 version.

HRT-HOOD is not the sole method that can claim advanced support for real-time system design.

Cornwell [15] proposes a methodology for component-based hard real-time system development consisting of a process model, which extends the TARDIS framework [16] and a method largely based on HOOD. Although predating the UML, Cornwell's work is relevant to our project, since it also addresses component-based development under HOOD. Cornwell combines HRT-HOOD with Z [20] for specifying object functionality, and RTL [19] for the expression of timing constraints. He also extends the HOOD notation to provide an explicit required interface specification as part of the object representation, and a clearer differentiation between classes and objects. Cornell's work also uses the notion of placeholder objects for composability, but in a context vastly different from ours. Cornell's placeholder objects are used to defer implementation of an object (whereby they only consist of a provided and required interface) in the context of the so-called *vertical interconnection*. Conversely, our placeholder objects (discussed in the following section) loosely correspond to the required object interface and are used in the context of Cornell's *horizontal interconnection*, i.e. the composition of objects at the same level of abstraction.

Another well-known method is ROOM [11]. In ROOM, *actors* are the main structuring abstraction. Actors are encapsulated units that model active system components and can be decomposed into contained actors. Actors can be derived by inheritance from other actors. Furthermore, an actor's interface is defined by a set of *ports* (as opposed to by operations as in HOOD), which model communication terminals.

Accordingly, a port carries both structural information on the connection between actors and *protocol* information that specify what messages can be exchanged across the connection. A state machine and/or a message sequence chart (similar to UML sequence diagram) may be associated to a protocol to express the allowable message exchanges.

Two actors can interact if there is a *binding* (an abstraction of a communication connection) between any two ports that the actors own and that support the same protocol in complementary (or *conjugated* in ROOM speak) roles. An actor's behaviour is specified by means of a ROOMchart, a form of statechart.

The ROOM design abstractions have been shown to be compatible with the UML meta-model, wherein they can be represented by means of a small set of new *stereotypes*. In the UML version of the method, ROOM actors (renamed *capsules*, presumably to avoid conflict with standard UML actors) and ports are stereotyped classes, whereas protocols are stereotyped collaborations. The structural role of ports shows from the choice of modeling them as classes (or class instances) rather than interfaces: UML interfaces can only specify behaviour in the form of a list of operations accepted by classes that realise an association; a ROOM port also specifies that the operations are valid in the context of a specific association. Ports enable designers to state explicitly which pairs of objects are partners in a given communication relationship and under which protocol the communication takes place. A collaboration is thus the appropriate UML construct to model a ROOM protocol and a port is represented as a capsule component that realises a role in the collaboration that models the port protocol.

Interestingly, no specific notations have been proposed, in either plain ROOM or the ROOM UML extension, to express timeliness-related properties and constraints.

COMET [6] is another real-time design method of interest, which from its outset relies on a set of UML stereotypes. COMET offers a larger assortment of abstractions than ROOM. The system structure is described in terms of classes stereotyped according to the role of the component they model. For example, COMET includes stereotypes for input and output interfaces, for coordinator objects, and for data. Collaboration diagrams and statecharts are used to specify behaviour. Some items of timeliness-related information such as activity period and computation time can be expressed as annotations to sequence diagrams, but the proposed annotations do not seem to correspond to any formal extension of the UML.

Arguably, the common underlying goal of ROOM and COMET is to seek an acceptable compromise between the appeal of generality and the urge for specialisation, the former being the toll to pay to commercial fortune.

The distinctive character of HRT-HOOD was to privilege specialisation downright. Which choice, we contend, had two contrasting consequences. The first, commercially negative, was to restrain the spread of the method to the limited market penetration of HOOD base technology. The second, especially beneficial in retrospective, was to limit the scope and the complexity of the method, with positive repercussions on the user's learning curve and the comparative ease of transposal into a tailored UML profile for the sake of preserving its methodological value from the decline of the HOOD-based technology.

Before delving further into the core matter of this paper, we wish to position our objective with respect to other research on schedulability analysis of UML models and to the recent response to the OMG Request for Profile for Schedulability, Performance and Time (cf. [12]).

Schedulability of UML models is discussed in [18] and [17], which both release the requirement (common to HOOD and other hard real-time design methodologies) of associating a thread to each active object and also allow active objects to exchange multiple synchronization signals in a controlled way. MAST [17] promotes a new UML profile for schedulability of objects mapped onto fixed-priority threads. Schedulability analysis of UML models in [18] uses the so-called event-based design paradigm, where priorities are associated to events and inherited by the handling objects and the threads implementing their actions.

The November 2001 response to the OMG call defines a comprehensive conceptual framework that extends the UML meta-model and which inevitably has a much broader scope and a higher abstraction level than any real-time design method proposed thus far. Our HRT-UML profile, in contrast, aims at a much less ambitious goal. It can be expected, though, that any new analysis and design methods in the real-time domain (including ours) should take benefit for the emerging OMG framework. Expectedly in fact, method-specific concepts could be defined more clearly, more rigorously and more portably if expressed in terms of the more general OMG framework. Standard ways to express timeliness and quality of service attributes could be reused. Existing UML technology could be extended to support specific methods with less effort.

3. Mapping Issues

The mapping strategy we have adopted rests on two founding principles: (**i**) preservation and reinforcement of the HRT-HOOD development model, which constructively steers the design process within the boundaries of the computational model of choice, thereby leading to systems that can be statically analysed from the outset; (in effect, as discussed in [14], HRT-HOOD fosters the execution of a "reflective" design process that trades exceeding expressive power and design freedom for structural conformance to a rigorously-defined computational model — i.e. a specific set of objects, relations, operations and semantics — from analysis throughout to coding); (**ii**) transposition of (dated) HOOD and HRT-based concepts and notations into UML, by refinement, clarification and replacement, following what we may regard as a process of UML rendering and augmentation of the [HRT-]HOOD meta-model.

This strategy yielded quite an interesting negotiation between the pull for conservation and the push for innovation exercised by the two competing forces. In the following we illustrate some of the major findings that we have come across as part of this process. In particular, we will focus our discussion on three main aspects of our reflection: (**i**) the prevalence of objects over classes (and of the object model over the class model) in an HRT-minded object-oriented development; (**ii**) the opportunity to derive classes by means of object-level examples and, thus, of feeding and relying on an "immanent" class model, which exists in the background without getting in the way of the HRT designer; (**iii**) the appeal of composable objects, which allow the designer to differentiate between "required" objects and "offered" instances and to replace the former with the latter as the development progresses.

3.1. The Prevalence of Objects

The wisdom of standard object-oriented development is that the class model be the main constructive model and the focus of development. Object modeling in this context configures as a supplementary activity intended to clarify aspects of behaviour and collaboration among class instances. The scenarios and the collaboration diagrams that are used for this clarification purpose neither reveal nor imply the way the system is built; they merely provide an aid for the analyst to better understand and convey the system behaviour in some specific circumstances.

The HRT-HOOD development philosophy, instead, requires all system components to be statically known in number, operations and relations, and be fully characterised for their real-time attributes, at design time. From an object-oriented standpoint, therefore, this vision does promote objects to first-level development entities over and above classes.

More specifically, the defining information for an HRT-HOOD system is the *topology* of its objects, that is to say, what the objects are in HRT terms and the way they are interconnected to one another, which HOOD describes in terms of *use* and *include* relationships. An HRT-HOOD system is thus built by directly defining its objects and their links, instead of its classes and their relations.

In fact, as we have seen, the object-basedness of HOOD actually degrades classes to *singletons*, that is classes with a single instance. This approach collapses the class model into an object model and attains an awkward isomorphism between the two, which UML is not really able to express, given the difference it purposefully sets between the respective notations and semantics.

On this and other accounts that we discuss below, we regard the isomorphism that HRT-HOOD imports from HOOD as neither useful nor especially desirable to HRT design.

HOOD concentrates on the object model solely and provides very modest support for class-level modeling. UML, conversely, brings class modeling to the foreground, but with semantics that are plainly inadequate (because not intended) for expressing instance properties. Not surprisingly, therefore, this differing strategies complicate finding a useful semantic mapping between models expressed in the two notations.

Our view of HRT design sides neither with UML nor with HOOD.

We contend that HRT design requires the object model to come to the foreground and the class model to rest in the background, and does not necessarily want them to be isomorphic. In fact, an HRT object-oriented design may arguably gain from the ability to express multiple static instances of well-formed HRT objects. A simple example will illustrate the apparent need for this provision and the power of the corresponding concept.

Before discussing the example, though, we should note that, by dropping the HOOD constraint of singleton-based isomorphism between class model and object model, we have effectively taken a middle ground between pure object orientation (as typified by the UML meta-model) and pure HRT-HOOD (as dominated by the HOOD meta-model). Table 1 positions our approach with respect to the two pure alternatives.

	Normal OO development	**HRT-UML development**	**Singleton HRT-HOOD**
Class model	main constructive model	background concept	isomorphic
Object model	analysis aid	main constructive model	

Table 1. Development paradigms vs. class model and object model.

3.1.1 Multiple Static Instances and the Underlying Class Model

The HRT-HOOD development paradigm that we want to preserve requires all systems components (all objects) to be fully characterised for type, attributes, operations and relations, at design time. This requirement clearly excludes the use of any form of dynamic creation of instances that occur at run time, which is one of the outstanding features of pure object orientation. At the same time, the HOOD meta-model concept of singleton class disallows even the very concept of multiple instantiation.

Allowing static (design-time) multiple instantiation is thus the obvious middle ground between dynamic run-time creation and plain single instances. The question is whether this concept fits the HRT development paradigm methodologically and is at all useful for it. We contend that it does and that it also reinforces our argument for bringing the object model in the foreground and for keeping the class model present but in the background.

Consider a simple real-time system comprised of two light bulbs (a higher-powered, primary one, and a lower-powered backup one), two thermal threshold sensors (one placed near the primary bulb, on the surface plane of the system and another, near the backup bulb, below the system surface) and two coolers that are primed into service by a dedicated driver connected to a specific sensor. The system is designed to operate with the primary bulb and its sensor-cooler chain on, except when the system temperature is sensed to exceed an alarm threshold that the cooler cannot safely decrease without switching the primary, more energy-dissipating, bulb off and switching the backup one on in its place until the system is safe again. The two sensors are independent of one another and operate on different thresholds, as they monitor the temperature in the vicinity of their specific bulb of competence. The two coolers (and the respective drivers) are also placed in the vicinity of their respective driver, so that the sensor closer to the primary bulb commands into service the cooler in that zone, whereas the sensor nearby the backup bulb commands its own cooler (which however gets into service only when the backup bulb comes up).

This contrived example serves a three-fold purpose:

1. It makes a case for multiple static instances.

2. It manifests the inadequacy of the class model for the HRT designers to express and control the topological aspects of the system, which can only be suitably conveyed by object modeling.

3. It reveals the need for methodological mechanisms that enable the HRT designer to concentrate on object modeling while relying on an *underlying class model* to perform design-time multiple instantiation as well as assignment of specific attributes to the target instance.

Contention 1 clearly follows from the presence of duplicate instances of every component of the example system (which, in fact, form a two-redundant system), each with specific attributes, operations and relations.

Figure 1 sustains contention 2 by showing a UML class model for the example, whose absolute absence of topological information makes it unfit, because uninformative, for an HRT designer.

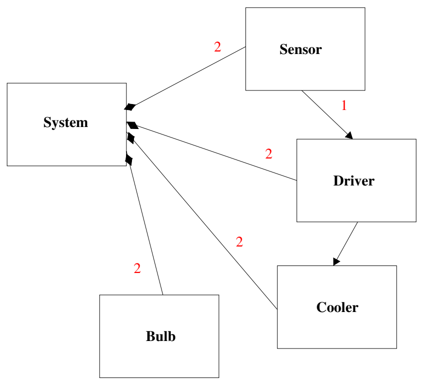

Figure 1. The lack of topological information in the class model for the example system.

The topological information of interest for an HRT designer would in this case include: which sensors is connected to which driver and to which cooler; what are the actual values for the characterising attributes of each object instance.

Contention 3 follows from the observation of how best equipped an object-model diagram like the one shown in figure 2 be to convey the required information.

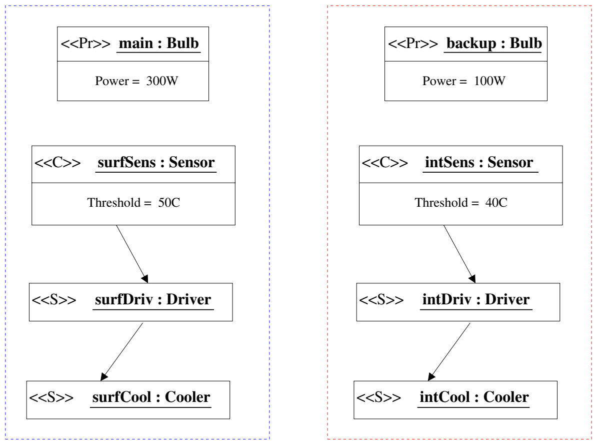

Figure 2. An initial object model that conveys topological information. The dashed boxes enclose objects and relations that can be viewed as instances. HRT-type tagging of objects uses the UML stereotype mechanism.

The highlight of this example, as captured in table 1, is that we operate an inversion of the classical object-oriented development paradigm. In our view, in fact, the HRT designer starts out with object modeling, where he/she can express system topology and capture the need (the opportunity) for multiple static instances of objects initially drawn as singleton. This opportunity is apparent in the example system, where the dashed rectangles enclose objects and relations that can be viewed as (design-time, characterised) instances of an *underlying class model*, effectively inferred from the object model. We contend, in essence, that in our design paradigm the class model *proceeds* from, as opposed to precedes, the object model.

Notably, neither the Generic Object concept of HOOD v4.0 [8] nor its misleadingly-named "class" object predecessor of HOOD v3.1, whose aim was to enable multiple instantiation, are of any use to our design paradigm, because:

- generic objects are defined externally to the system under construction, and thus invisibly to the system objects, with which they can have no relationship;
- the instantiation of a generic object hides its internal decomposition structure (what we call its topology).

The irreducible difference between the HOOD v4.0 concept and ours is that we draw a clear separation between the class-model view of instantiatable objects (the generic object of HOOD v4.0) and the object-model view of concrete instances, which we treat as fully topologically characterised objects.We must now find a useful role and an adequate notational rendering for (parts of) the class model shown in figure 1, whose existence in the background is methodologically crucial to enable multiple instantiations.

The following section discusses in more depth the role we have assigned to objects and classes in HRT-UML.

3.2. Deriving Classes by Examples

The "OO prototype" design pattern proposed by the "Gang of Four" [5] promotes the idea that new object instances may be created by cloning of the structure defined by what amounts to the *prototype instance* of a class, that is to say, a typical object instance, initially drawn as a singleton. This is admittedly not a standard UML concept, but it very closely matches the design process we want to capture. (This notwithstanding, the HRT-UML method can be wholly expressed by way of standard UML constructs, with specialised support only required for the consistency checks of the user design.)

It is interesting to note that the intertwined notions of underlying class model and prototype instance hold at all levels of design hierarchy. (Hierarchical decomposition as a fundamental aid to mastering complexity is an area in which HOOD appears to be stronger than UML. HOOD calls parent an object that is internally decomposed, and terminal an object that features no internal decomposition. HRT-UML includes support for this design dimension, yet bearing in mind that the verification-geared nature of HRT development only tolerates forms of hierarchical decomposition that allow immediate visibility and early characterisation of design components.)

As the example system shows, we can identify prototype instances worth replicating at the level of individual objects (e.g.: bulb, sensor, driver, cooler) as well as at any aggregate level of them, up to the full chain of related objects enclosed in the dashed rectangles in figure 2.

The way replication (cloning) of such object occurs entails escaping implicitly from the object model view to the background call model view. To allow this escape, any object in an HRT-UML design carries a *descriptor*, which we can initially express as a pair comprised of: the underlying class, which describes the abstract properties of object instances of that class (e.g.: HRT type, provided operation types, required attributes); and the prototype instance, which describes the internal topology of the object, which must be adhered to when cloning the object. The former component would be expressed in class-model rendering and semantics; the latter in object-model rendering and semantics.

No components of the object descriptor would be displayed as part of the system design: they exist on a different plane, which we want to keep sharply separate from the design level. Notwithstanding this separation, we wish to allow the designer to escape to the object descriptor viewpoint, for example to study (for possible cloning) the prototype instance of a particular object, whether simple or aggregate. Representing entities in this viewpoint requires a specific notational rendering, which figure 3 illustrates.

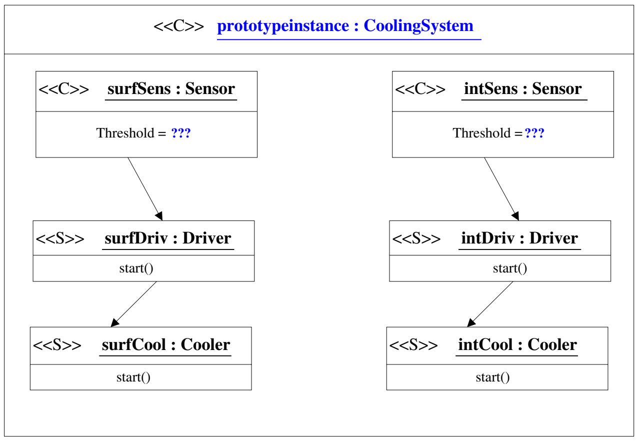

Figure 3. A notational rendering of the prototype instance of an aggregate subset of the example system, which shows the predefined operation compartment of sporadic objects.

3.3. Towards Composable Objects

The information value carried by the prototype instance part of the object descriptor needs to be explored further.

A crucial aspect of the definition of an object is what HOOD captures under the *use* and the *include* relationships. The former element may be regarded as the specification of the *required interface* of the object. The latter as the implementation of the *provided interface* of the object.

From an object-model perspective, both interfaces have to be fully defined and fully characterised (hence, be realised) for a system design to qualify as HRT. For example, the provided interface of an HRT-HOOD sporadic object must include an asynchronous start operation that deliver the triggering event for the object's thread of control. Likewise, the required interface of an HRT-HOOD protected object cannot include any potentially blocking operation.

From an object descriptor standpoint, however, such interfaces not only can be concrete (i.e. directed towards specific object instances), but they can also simply express requirements as to what would be required of internal and external object instances to satisfy the interface specification of the object in question.

We use the notion of *placeholder object* to capture this concept. A placeholder object in the prototype instance of an object descriptor specifies the interface requirements to be satisfied for an instantiation of the aggregate object to succeed. The prototype instance concept promotes the use of interfaces instead of classes as target entities of associations. This reflects the fact that interface specification (which bases on the corresponding UML notion, but also includes all the information contents required for HRT modeling) is more productive and expressive for our paradigm than plain classification.

We can illustrate the use of the placeholder concept as we look back at the example system.

Assume, for example, that we wanted to bring forward the prototype instance of a Sensor object. Information extracted from the object model shown in figure 2 tells us that, for the purposes of the system under construction, objects of class sensor carry an association (a link) to objects of class Driver. The prototype instance of the Sensor object descriptor expresses this notion by exhibiting a link to a placeholder object of type Driver.

Placeholder objects are initially loose (i.e. unbound) when the prototype instance is created. Such creation occurs implicitly in the background dimension where the object descriptor resides. In the foreground design dimension, in the course of the development process, placeholder objects are bound to actual object instances that are visible in the instantiation environment and that satisfy the interface requirements.

The notion of placeholder object enables the HRT development paradigm to accommodate the use of the "builder metaphor", so successful in GUI development, in the way, for example, of Visual Basic, JBuilder, Visual Age, and the like.

By use of placeholder objects, HRT development could tread the path of reuse, by designing (parts of) the system as the assembly of static instances of classes that satisfy specific requirements.

The expressive power of placeholder objects is especially attractive for reuse-geared, component-based HRT development. Placeholder objects may be the target of both *use* and *implemented by* relationships, which HOOD and HRT-HOOD employ to model the realisation of, respectively, required and provided interfaces, and which we preserve in HRT-UML. Thanks to this ability and the combined use of prototype instance and hierarchical decomposition, we can allow two flavours of HRT component: one in which the link to the placeholder object is external to the prototype instance in question (which thus denotes a fully-resolved component that allows multiple – but complying – instantiation contexts); another in which the prototype instance itself is a composite, non-terminal object whose provided interface may be realised (implemented by) compliant instantiations of specific placeholder objects. The latter flavour permits full-fledged forms of reuse by adaptation and configuration. As an example, we maintain that the OBOSS library of domain-specific reusable HRT components, described in [13], is especially suited for being expressed in HRT-UML by use of placeholder objects. We plan to explore this issue further in future work, as HRT-minded reuse is an intriguing area of research.

4. Conclusion

Attempting to map HRT-HOOD onto the object-oriented meta-model of UML raises methodological issues that concerns the real-time design process in general. In this paper we have discussed what we believe are the major issues we encountered in the definition of HRT-UML as the receptor of the unique strengths of the HRT-HOOD method expressed in terms of UML. By means of a simple example and some conjectures that need to be consolidated by specific case studies (that we plan to execute in due course), we have shown that: (1) real-time design requires the object-model view of the system to come to the foreground, thus relegating the class-model view of the same to the background; (2) the need to effectively support multiple, design-time, instantiation, which we consider useful and beneficial to real-time design, calls for the ability to derive classes "by example" from individual objects existing in the object-model dimension of the design; (3) the use of UML interfaces, augmented with HRT-HOOD derived attributes, as the target of associations (links) among objects allows attractive forms of component-based real-time minded development. These three dimensions were inhibited to HRT-HOOD by the object-based nature of the underlying HOOD meta-model.

The greater expressive power afforded by the transposition onto UML has loosened the restraints exclusively due to HOOD. The result is an augmented HRT development paradigm that, in our expectation, will preserve the methodological strengths of HRT-HOOD beyond the commercial decline of HOOD technology.

References

[1] A. Burns. The Ravenscar Profile. *Ada Letters*, XIX(4):49–52, 1999. ACM Press.

[2] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, University of York, 2003. `http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf`.

[3] A. Burns and A. Wellings. HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. *Real-Time Systems*, 6:73–114, 1994. Kluwer Academic Publishers.

[4] A. Burns and A. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Systems*. Elsevier Science, Amsterdam, NL, 1995. ISBN 0-444-82164-3.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissideds. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1999.

[6] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000. ISBN 0201657937.

[7] HOOD Technical Group. *HOOD Reference Manual 3.1*. Prentice Hall, Englewood Cliffs, NJ (USA), 1993.

[8] HOOD Users' Group and J. Rosen. *HOOD - An Industrial Approach for Software Design*. HOOD Users' Group, 1997. ISBN 2-9600151-0-X.

[9] ISO. *Ada Reference Manual*. International Standardisation Organisation, Geneva, CH, 1987. ISO/IEC 8652:1987.

[10] Revision Task Force. OMG Unified Modeling Language Specification, Version 1.4. Technical report, Object Management Group (OMG), 2001. OMG document formal/01-09-67.

[11] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. Wiley & Sons, 1994. ISBN 0471599-174.

[12] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational, `http://www.rational.com/products/whitepapers/umlrt.pdf`, 2002.

[13] T. Vardanega and G. Caspersen. Engineering Reuse for On-board Embedded Real-Time Systems. *Software - Practice and Experience*, 32(3):233–264, 2002.

[14] T. Vardanega and J. van Katwijk. A Software Process for the Construction of Predictable On-Board Embedded Real-Time Systems. *Software - Practice and Experience*, 29(3):235–266, 1999.

[15] P.D. Cornwell. Reusable Component Engineering for Hard Real-Time Systems. *DPhil Thesis Department of Computer Science. University of York*, 1998. `http://www.cs.york.ac.uk/ftpdir/reports/YCST-98-04.ps.gz`

[16] A. Burns and A Lister. A Framework for Building Dependable Systems. *Computer Journal*, 2(23):173-181, 1990.

[17] J.L. Medina, M. González Harbour, and J.M. Drake. MAST Real-Time View: A Graphic UML Tool for Modeling Object-Oriented Real-Time Systems . *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, London, UK, IEEE Computer Society Press, pp. 245-256, December 2001.

[18] M. Saksena, A. Ptak, P. Freedman, P. Rodziewicz. Schedulability Analysis for Automated Implementations of Real-Time Object-Oriented Models. em Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998), Madrid, Spain, December 1998.

[19] F. Jahanian and A. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions in Software Engineering*, SE-12(9) 1986.

[20] R. Di Giovanni and P.L. Iachini. HOOD and Z for the Development of Complex Software Systems. *LNCS(428)*:262–289. Springer Verlag. 1990.