Automated Model-based Generation of Ravenscar-compliant Source Code

Matteo Bordin, Tullio Vardanega
Department of Pure and Applied Mathematics, University of Padua
via G. Belzoni 7, I-35131 Padova, Italy
mbordin@studenti.math.unipd.it, tullio.vardanega@math.unipd.it

Abstract

Graphical languages of various sorts are increasingly used for the specification and the design of high-integrity real-time systems. Their coverage however does not extend with as much success to automated source code generation. Several hurdles cause the model-to-code translation to often lapse in the preservation of the desired semantics. This paper illustrates the choices we have made to provide the HRT-UML design method with an automated Ravenscar-compliant source code generation engine. Compliance with the Ravenscar computational model warrants static analysability of the source code and predictability of execution. By elevating this compliance to the design stage, we earn semantic preservation across the whole development process.

1. Introduction

Observation of current industrial practice shows that various brands of graphical languages are increasingly used for the specification and the design of high-integrity real-time (HRT) software systems. Equally visible evidence however indicates that very little reliance is placed on the automated source generation capabilities of such infrastructures.

Ideally, one would regard software design as a foreground primary activity, largely disentangled from background software coding. The former could thus concentrate on the semantic aspects of value to the designer and be abstracted away from irrelevant syntactic details. For this view to hold however, the translation from design to code must of course be free from semantic breaches.

Manual coding is intrinsically exposed to this risk, especially so in the face of the increasing dimension and complexity of modern HRT systems. The alternate route, the use of automated source code generators, though obvious, has so far proved somewhat impervious. Several technical and strategic difficulties make automated source code generation a rather arduous task.

Automated source code generation should fit in pre-existing industrial processes, and should make them evolve without breaking. Moreover, it should co-exist with domain-specific legacy, which may range from simple idioms to vast libraries of proven code, and yet be able to consistently accommodate it without loss of integrity. Furthermore and paramount, it should fully and completely adhere to the computational model used for the creation and the analysis of the design.

The computational models for HRT systems are very restrictive, because they are meant to warrant static, i.e. design- and compile-time, verification of the properties of interest to the user (notably, space and time predictability of execution). In the HRT domain, therefore, by validating the design model one wants to acquire absolute certainties on the run-time behaviour of the corresponding code. This demand entails the preservation of a two-staged correspondence: (1) from model to source code; (2) from source code to executable. A single leap, directly from model to executable code, is presently not foreseeable because of inherent limitations in the expressive power and functional coverage of the modeling technology.

To serve larger segments of users, mainstream programming languages tend to support rich and permissive computational models.

Some languages intentionally abide by no computational model at all and defer the choice to the language binding to an underlying operating system. This choice is maximally liberal, but it exacerbates the problem, for it adds an extra step, and a critical one at that, to the verification chain.

Other languages (Ada [1] and the Real-Time Specification for Java [11] chiefly) do embed a variety of concurrent computational models, but not necessarily come to enforcing any of them.

Yet, compile-time enforcement crucially adds to the integrity of the final software product, since automated source code generation hardly eliminates the need for the insertion of foreign (possibly manual) code in the production process.

One computational model of considerable interest to HRT systems is the Ravenscar Profile [5], a subset of the

Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)
0-7695-2400-1/05 $20.00 **IEEE**

Ada 95 tasking model [1] especially tailored for static verification. The latest revision of the language due for late 2005 [2] has fully endorsed the notion of *profile*, so that a program that chooses the Ravenscar Profile (and any other supported one) will be checked for compliance by both the compilation system and the run time system.

The Ravenscar Profile is based on fixed-priority preemptive scheduling [3, 4] compounded with priority ceiling emulation [10, 19] to optimally bound priority inversion, to achieve lock-free mutual exclusion and to also avoid deadlocks.

The profile places a number of tight restrictions on the native wealth of Ada concurrency constructs. As a result, it defines a concurrent computational model that renders its applications structurally amenable to static analysis and allows for highly efficient, reliable and certifiable implementations of the underlying kernel [22].

The Ravenscar computational model thus is an excellent target for automated source code generation.

The Real-Time Specification for Java [11] has assumed a number of characters that can be paralleled with the Ravenscar Profile and spawned proposals for language-level profiles (cf. e.g.: [12]). It may thus be hoped that adherence to a single computational model may not impose the choice for a single programming language.

This paper illustrates the choices we made in equipping the HRT-UML design method with capabilities for automated Ravenscar-compliant source code generation.

HRT-UML [13, 21] is the result of a project that is to transpose, by a thorough balance of preservation and extension, the HRT-HOOD design method [6] on the UML meta-model [15, 14]. A prototype tool-set currently exists, which we have equipped with the proposed code generation engine.

The original HRT-HOOD method definition included mapping guidelines for automated code generation of Ada code. Those guidelines however were defined long before the Ravenscar Profile came to be, and therefore without the need to warrant structural compliance with the profile restrictions.

As a result, although the original guidelines aimed at producing code that could be statically analysed for schedulability, that code would fail to compile under the Ada 2005 check for compliance with the Ravenscar Profile.

The contribution of this paper is to discuss the implications of Ravenscar-compliant code generation on the design method itself, so that the user of HRT-UML is offered, in effect, a consistent *design profile* and not just a suite of mapping guidelines that bend and twist the design model semantics to fit a specific programming style. The former choice caters for round-trip engineering (the ability to keep design models and code in sync with one another), an essential asset for generative programming [7] and model-driven architecture [16] approaches. The latter obviously does not.

2. HRT-UML and the Ravenscar Profile

HRT-UML retains the hierarchical nature of the base HRT-HOOD method so that its design objects may be either parent (i.e. internally decomposed) or terminal. Terminal objects must hold a given HRT type. The common abstractions for HRT design were first characterised by HRT-HOOD in a manner that revolved around static (i.e. design-time) association between objects and flows of control. Other characterisations are obviously possible, such as e.g.: in ROOM [18] and COMET [9]. What makes our preference go to the HRT-HOOD one, though, is precisely its distinct strategy of association.

We briefly recall the basic abstractions for terminal objects, noting that the HRT-HOOD meta-model semantics prescribe that objects may provide public operations (i.e. methods) that may be invoked as a result of a direct or indirect calls made by the threads of some object.

PASSIVE objects have no control over when invocations of their operations are serviced and do not spontaneously invoke operations in other objects. PASSIVE objects raise no concurrency issue and therefore are not impacted by the Ravenscar Profile restrictions. As such they are not discussed any further in this paper.

PROTECTED objects may have control over when invocations of their operations are serviced and do not spontaneously invoke operations in other objects. The control conditions allowable for the callee must warrant bounded wait time on the callers.

CYCLIC objects represent periodic threaded activities and may spontaneously invoke operations in other objects. The operations that CYCLIC objects are allowed to invoke must be unconstrained.

SPORADIC objects represent sporadic threaded activities and may spontaneously invoke operations in other objects. SPORADIC objects are allowed to invoke a single potentially blocking operation and any number unconstrained ones. The single potentially blocking operation must be the one that delivers the activation event for the object's thread of control. (Invocation constraints and blocking conditions are discussed in a separate subsection.)

HRT-HOOD also allowed for ACTIVE objects, to denote objects with an own thread of control, and therefore similar in nature to CYCLIC and SPORADIC, but deprived of the structural constraints applied to them. HRT-UML deprecates the ACTIVE type, for it leads to objects that do not warrant static analysis and therefore defeat the purpose of the method.

Objects are described by their provided and required interfaces, whilst their internals are hidden from the view of users. The (hidden) internals determine the value of the

HRT attributes that characterise the object externally. The internals of an object are made up of three distinct parts:

- an agent that implements the invocation constraints associated with the provided interface of the object; the agent is named OBCS (after object control structure); the invocation constraints determine when and how the incoming requests for interface operations published by the object may be serviced: the job of the OBCS is to realize the relevant invocation semantics; the presence of an invocation constraint in the provided interface of an object requires an OBCS in the implementation of that object; invocation constraints may defer servicing of the call until functional or non-functional (i.e. synchronisation related) conditions hold;

- the concurrent activity within objects that possess an own flow of control, which is encapsulated in a threaded entity notionally named THREAD, which is the entity effectively executing the required service actions in response to accepted invocation requests;

- an entity, named OPCS (for operation control structure), which encloses the implementation details of all the object operations, in conformance with the good old information hiding principle.

In effect, the realisation of any constrained operations in the provided interface of an object is placed under the responsibility of the object's OBCS and of its interactions with the object's OPCS as executed by the object's THREAD. All of this happens invisibly from the caller's view. It is the mediation of the OBCS that allows for concurrent execution in the caller and in the callee objects (which is not warranted, instead, by the sole presence of the OPCS).

The goal of the code generation engine is to *fully* automate the generation of the source code for the OBCS and THREAD components (only allowing user parameterisation of object-level attributes, such as e.g.: priority, importance, initialisation values).

Our project goal is that user intervention should be limited to providing source code for the OPCS, which is the functional component of the object.

Invocation Constraints. Constraints on the invocation of interface operations need to be characterised in two respects: on the way their servicing may depend on the internal state of the callee (which the design method limits to the classical form of avoidance synchronisation, and which is historically called *functional activation constraint*); and on the synchronisation effect that the invocation may have on the caller (this is historically called *request type* and it is viewed as a static attribute of the operation and not of the call). The request type might take one of three forms:

- *asynchronous* execution request (ASER), which entails no suspension of the caller;

- *loosely synchronous* execution request (LSER), which causes the caller to suspend until the callee is ready to service the invocation;

- *highly synchronous* execution request (HSER), which suspends the caller until the callee has serviced the request.

The HSER and LSER variants have been expunged from HRT-UML, for they entail synchronisation between the threads of the caller and the callee, which is not amenable to static timing analysis. No provided interface of HRT objects may therefore include HSER or LSER operations. The rationale for this decision stems from the Ravenscar Profile restriction [5] that prohibits any form of synchronisation between threads of control. In fact, this restriction forms the very reason for the introduction of PROTECTED objects as the sole means for threads to interact with one another.

In HRT systems, however, a further variant of invocation request must exist, to cater for protected access to shared data. This may have an associated functional activation constraint, and therefore be *protected state-constrained* (PSER), or else have none, thereby becoming the "protected" analogous of the ASER, hence called *protected asynchronous* (PAER). The former variant provides for data oriented synchronisation, whereby two cooperating objects may asynchronously exchange data and state information in a form that warrants a prescribed ordering of execution and mutually exclusive access to shared data (e.g.: PAER "set", with no functional activation constraint, which opens the constrained PSER "get").

Constrained interface operations must exhibit a specific request type, while may have a functional activation constraint. A caller awaiting for a functional activation constraint to be asserted is said to be *blocked*, while one that simply awaits for a synchronisation is said to be *suspended*. Invocations that are subject to functional activation constraints are therefore potentially blocking. Those that merely involve synchronisation are potentially suspending. In the absence of structural restrictions, both the blocking time and the suspension time may in principle be unbounded. Bounding may in principle be obtained by setting a time-out on the corresponding invocations and by tagging the relevant request type with a TO_ prefix. Note however that asynchronous time-outs break the nominal logic of the application, thereby incurring non-deterministic execution, much against the intent of the Ravenscar Profile.

The Ravenscar version of HRT-UML does not allow the OPCS of threaded objects to execute potentially suspending invocations as they break the prerequisite for static analysis. We readily enforce this prohibition by banning HSER and LSER from provided and required interfaces altogether.

HRT-HOOD also wanted parent objects to be attached an HRT type at the time of creation, which would prescribe the allowable decomposition for the object. The intent was that non-terminal threaded objects (i.e. CYCLIC or SPORADIC) would represent cyclic or sporadic *transactions*, in the form of a precedence-constrained set of cooperating child objects. In fact, use experience has shown that real-life design does not rigidly follow the top-down approach entailed by hierarchical decomposition, but it has very distinct bottom-up connotations, whereby objects are also aggregated *a posteriori*. This notion, which HRT-UML promotes [13], suggests that the HRT type of a parent object is determined by (as opposed to determines) the HRT type of its child objects. Any aggregate of HRT-UML objects may be defined at design time as a single-parent by promoting (segments of) the provided interface of its internal objects to the parent interface. The resulting composition of the parent interface is then statically checked for legality and the HRT type of the parent object determined accordingly.

An HRT-UML object is fully represented by its provided and required interfaces. Each such interface thus plays the role of a placeholder. A placeholder fully specifies the constraints placed on the invocation of the interface operations. It therefore follows that the bottom-up creation of a parent object may be aimed at matching a placeholder interface. A parent in this case would actually be a component aggregate, a possible candidate for reuse.

As code is generated for fully instantiated systems (in which all placeholders have been replaced by concrete objects), discussion of automated code generation for placeholder interfaces falls outside the scope of this paper.

3. Code Generation for Terminal Objects

3.1. PROTECTED Objects

The intent of PROTECTED objects in HRT-UML has stayed much the same as it was in HRT-HOOD. They are used to provide controlled access to resources that are in shared use by other objects.

Ravenscar PROTECTED objects must warrant bounded blocking time for the caller of any PSER operations. Furthermore, PROTECTED operations must be designed so as to minimise the irreducible duration of priority inversion caused by mutual exclusion. This is obtained by preventing the OPCS of PROTECTED objects from invoking operations marked as either potentially blocking or potentially suspending.

The former prohibition can easily be enforced at modeling level: no PROTECTED object can have a use relation directed toward a PSER operation in any placeholder or concrete provided interface. (The reader should note that PAER operations pose no problem in this regard, for the ceiling locking policy in force with the Ravenscar Profile ensures that no PAER invocation may ever incur suspension [22], while they do not incur blocking by definition.)

The latter prohibition instead requires syntactic verification that the functional code attributed to the OPCS does not contain self-suspending statements (delay [until] in Ada and sleep in (RTS) Java). This verification is eventually carried out by the Ravenscar-mode compiler for Ada, but it can easily be undertaken by the design tool itself by transitively applying the restriction to the OPCS of all objects. If the verification is deferred to the compiler, the restriction will hold only for the code directly and indirectly invoked by the OPCS of PROTECTED objects. Otherwise, the restriction will hold throughout the entire application, prohibiting the use of self-suspension in OPCS code *tout court*. It is for the user to decide which way to go in this regard.

A further Ravenscar restrictions holds on the number of PSER operations that can appear in the provided interface of PROTECTED objects. There can be only 1. Moreover, there can be no more than 1 caller for any single PSER interface operation in the system. The combined effect of these restrictions is that no queues of (blocked) threads can ever form on any interface operations. This dismisses the extent of non-determinism that arises from waiting in queues that hold multiple threads and from more than 1 state constraints becoming open simultaneously. Both restrictions can easily be enforced at modeling level. Notably, they also cure the flawed behaviour of the wait / notify primitives of Java [17].

As we saw in section 2, PROTECTED objects need an OBCS, for their provided interface may include PSER and / or PAER operations, which are in fact constrained operations. There is no need, though, that such objects also include a THREAD. Hence, in the interest of space and time optimisation, PROTECTED objects are not threaded. When it comes to automated code generation, then, the net consequence of this optimisation is that the OBCS *is* the PROTECTED object, whereby the former becomes subject to the same Ravenscar restrictions as the latter.

The following code generation rule therefore arises:

- the provided interface of a PROTECTED object is realised by delegating (i.e. mapping) *all* of its operations to an internal OBCS and by implementing them via internal invocation of the object's OPCS.

For obvious semantic reasons, the delegation must *not* create a run-time entity. Hence, the internal OBCS is, for all intents and purposes, *the* PROTECTED object. The restrictions placed on the object type ensure that this code generation rule preserves compliance with the Ravenscar Profile.

3.2. CYCLIC and SPORADIC Objects

Common issues. The Ravenscar restrictions hit the modeling of CYCLIC and SPORADIC objects only as regards

the implementation of ATC (Asynchronous Transfer of Control) operations.

The nature of CYCLIC and SPORADIC objects imposes strict limitations on the way they may receive notification of external events. For both objects, a relevant notification would for instance inform of a system-level error. For CYCLIC objects a further notification of importance would be a mode change request, which changes the period of execution.

It is possible but not very plausible that the OPCS of such objects should make explicit provisions for polling such notifications from the designated information sources, presumably embedded in PROTECTED objects. This approach has two downsides: the undesirable latency induced by polling; and the devolution to the user of the burden of programming the whole details of the notification. The latter defect defeats the intent of maximising the coverage of automated code generation.

In analogy with general execution requests, three basic forms of ATC invocation may appear in the provided interface of threaded objects: *asynchronous* (ASATC), which entails no suspension of the caller; *loosely synchronous* (LSATC), which suspends the caller until the OBCS of the callee acknowledges reception of the request; *highly synchronous* (HSATC), which suspends the caller until the callee has serviced the request.

Given the structure of threaded objects, the synchronous variants of the ATC invocation would synchronise the OPCS of the caller (and therefore its THREAD) with the OBCS of the callee (hence *not* with its THREAD).

In the interest of streamlining the automated source code generation logic, we subject the OBCS of threaded objects to much the same mapping strategy as we illustrated for PROTECTED objects. This factoring incurs no adverse semantic effects, because the Ravenscar Profile allows the OBCS structures and the PROTECTED objects to amount to the same meta-model semantics with bounded loss of generality and of expressive power.

To apply the same mapping strategy implies that the Ravenscar restrictions imposed on PROTECTED objects must hold for the OBCS of threaded objects as well. Consequently, the only form of synchronous request type that can appear in the provided interface of (what amounts to an internal) PROTECTED object is the PSER. The Ravenscar restrictions limit the number of PSER in a PROTECTED interface to 1 and also the number of system-wide callers of it to 1. If the provided interface of the SPORADIC or CYCLIC object were to include multiple synchronous ATC operations, then they would have to be mapped to as many internal PROTECTED objects, each playing the role of the object's OBCS, which is plainly unpractical. Furthermore, even if there was a single ATC operation in the provided interface of the object, then there should be no more than 1 caller for it. This caller could only be a SPORADIC THREAD, which should attach its single activation event (normally satisfied by the private PSER operation provided by its OBCS) to the synchronous ATC operation in question. This restriction would therefore prevent any form of centralised dispatching of synchronous ATC notifications for multiple objects.

On account of these two important limitations, the Ravenscar version of HRT-UML deprecates LSATC and HSATC, and only supports ASATC invocations.

Issues specific to CYCLIC objects. The realisation of ASATC invocations can only be mapped to PAER operations in the object-private provided interface of the OBCS. The OBCS is only a mediator agent and thus can undertake no service action on behalf of the object's THREAD. The structural restriction that a CYCLIC object can only invoke unconstrained operations does therefore apply to the object's THREAD and to its OPCS component.

Hence, it follows that the THREAD of the CYCLIC object can only invoke PAER operations in the object-private interface of its OBCS. Invoking a PAER operation is thus the *sole* means allowed for the CYCLIC THREAD to acquire the notification delivered by an ASATC called by an external object. In other words, the Ravenscar-compliant version of HRT-UML can only offer, for CYCLIC objects, polled acquisition of asynchronous notifications conveyed by ASATC invocations.

Overall, the provided interface of CYCLIC objects can include as many ASATC operations as the user wants (and this permission also holds for SPORADIC objects as well). Such operations will directly map to as many PAER invocations on the relevant object's OBCS. Invocation of the appropriate PAER will enqueue the ATC request descriptor in an internal protected queue of the OBCS. The CYCLIC THREAD will then periodically poll for any descriptor, fetching one among those that may have since accumulated, in accord with a configurable policy. The poll invocation will map to a PAER on the OBCS, which will return a null descriptor if there was no request to dequeue, so that the CYCLIC thread may then perform its nominal action.

Issues specific to SPORADIC objects. The handling of ATC operations is much simpler for SPORADIC objects. As we saw in section 2, the THREAD of SPORADIC objects can invoke a single potentially blocking operation. This operation is obviously a PSER and it necessarily appears in the provided interface of the object's OBCS. because it is from there (and not from the object's interface) that the THREAD will invoke it. A closed guard implementing the PSER functional activation constraint will block the THREAD's invocation until opened by an external caller of a PAER operation placed for this purpose in

the provided interface of the SPORADIC object. It follows that the provided interface of the object's OBCS includes one operation *more* (the PSER) than those (the PAER) that appear in the public provided interface of the object itself: The arrangement is thus that the public PAER delivers the triggering event for the SPORADIC THREAD, which receives it from the private PSER.

Multiple objects can invoke (possibly multiple) ASATC operations on threaded objects. The relevant invocations could easily be enqueued without affecting the caller and subsequently be fetched by the object's THREAD. Insertion and extraction could follow any suitable policy, FIFO being the default for any type of invocation. Other policies might be defined to order ATC invocations of differing nature (e.g.: mode change vs. error notification).

The nature of ATC invocations in HRT systems however is often that the newer call (of any given type) overrides the unattended older. Hence, our automated source code generation engine offers a default mode that uses a toggle buffer for holding the request descriptor conveyed by one type of ASATC call. The "write" position is overwritten by subsequent calls, whilst the "read" position is not. When the descriptor in the "read" position is read, the pointers are reversed so that the latest request may be read out. One fetch per activation of the object's THREAD occurs, from the most important ASATC queue as determined by the policy in force.

The provided interface of SPORADIC objects must also include an ASER operation that delivers the nominal activation event to the SPORADIC THREAD. That ASER will map to a PAER in the object's OBCS, which will add to the other PAER operations to which the object-level ASATC invocations are mapped. Each such PAER will open one and the same PSER invocation, which is only visible to the SPORADIC THREAD and on which the latter blocks while waiting for nominal or exceptional activation events. Exceptional activation events are fetched in precedence over nominal ones. The relative ordering of the former is determined as illustrated for CYCLIC objects.

4. Evaluation

4.1. Methodological Premise

We contend that a Ravenscar mode of HRT-UML design fully equipped with automated generation of compliant source code is comparatively easy to achieve. Other papers and reports (cf. e.g.: [8, 20]) also argue that the Ravenscar restrictions avail sufficient expressive power to the user. One aspect that needs assessment is the effect of the profile restrictions on the quality of the source code that can be possibly obtained by automated generation. The reader should note that we intentionally concern ourselves about the quality of the source (readable) code.

It is granted that the importance of space and time costs of the executable is paramount to HRT systems. Arguably, however, our structural compliance with the Ravenscar Profile earns us sufficient leverage to satisfy the needs in this regard. Yet, we contend that HRT systems are also reviewed at source code level. Usually they are ever more critically scrutinised if the source code is automatically generated. In fact, the trend is that the code generation rules are validated first and then the code generation engine itself. It is therefore of considerable importance that the product of the code generation rules be of high software engineering quality. In our view the criteria of interest in this regard would be: (1) *source code modularity*; (2) *source code factoring*; and (3) *space and time efficiency*. We will look at each in isolation.

The evaluation results we discuss have been obtained on the assessment of the Ada code generator. This generator engine is the most mature one, as work on the RTS Java code generator is still in progress, in part owing to the "in-progress" nature of some important aspects of the RTS Java semantics.

4.2. Source Code Modularity

The terms of reference we used for this assessment criterion is the one set by the source code resulting from the generation rules originally proposed for HRT-HOOD [6].

The original rules had several positive features (e.g.: the encapsulation of each design object in a self-contained package; the separation between OPCS and OBCS; the realisation of interface mapping by means of operation renaming). Our generation rules have preserved those assets and have compounded them with considerably greater attention to producing smaller-size and more cohesive units.

In fact, our generation rules break down each single design object into three distinct units, organised into a self-contained hierarchy (much in keeping with the same principles of hierarchical decomposition that inform the HRT-UML design method):

- the root of the object representation hierarchy (notionally denominated `Obj`) is a library level package that contains the object's provided interface, the object's internal attributes and the instantiation of the object's OBCS;
- a child of the root package (notionally denominated `Obj.Types`) contains the definition of the types and operations (the Ada equivalent of Java classes) which are internal, hence private, to the object operation; these operations are, in effect, the object's OPCS;
- another child of the root package (notionally denominated `Obj.RTAtt`) contains the real-time attributes of the

object, including the priority assigned to the object's thread of control (if any) and the period or minimum inter-arrival time of its activation.

Arguably, a decomposition that assigns the object internals to smaller-size, more cohesive, specific sub-units is more modular than one that gathers all the internals into a single source code unit.

Furthermore, the rules that govern visibility into hierarchies of packages warrant a much finer-grained extent of information hiding, which is also a bonus.

4.3. Source code factoring

Considerable benefit in this direction was obtained by recognising that the operation requests that client (caller) objects deliver to server (callee) objects are best represented as *reified* descriptors if source code factoring is to be maximised. In practice, we associate an explicit data object to every invocation request in the system, hence we reify them, so that we can have full control over how they are represented (in terms of an application specific derivation hierarchy) and thus on how they are handled at source code level.

Each request descriptor is comprised of: an enumerate value that identifies the request type in a statically-defined system-wide range; and a pointer to the specific parameters associated to the request instance, which are bundled together in a single record structure.

By building a single common set of operations that manipulate such request descriptors we have factored out a sizeable proportion of code that otherwise would have been repeated *ad-hoc* for each object.

In effect, we place an archetypal request descriptor and its basic handling operations (i.e. a class) at the root of a simple derivation hierarchy and define (and thus generate) the appropriate subclass of it for each object that needs to have its own operation parameters and its specific handling operations.

Static, compile-time, binding ensures that each OBCS executes the request handling operations that is specific of its pertaining object. To safely achieve this we simply have to equip the `Obj.Types` unit for the concerned object with the subclass required for the object's OBCS to handle the incoming operation requests. The net result of this measure is that our generator is able to factor out the *entire* source code for all types of THREAD and all types of OBCS, which achieves exactly what we set out for.

To leverage on this asset, we encapsulate those factored elements into generic templates and simply instantiate them as needed when we need to create object instances. In this way, we are able to reduce the specification of the `Obj` unit to simply declare the object's provided interface (hence add no code complexity whatsoever to the weight of the original design), while its body instantiates, where applicable, the object's THREAD and the relevant OBCS (which we bundle in a *single* generic template) and maps the corresponding operations.

Not surprisingly, the source code required for this purpose is extremely succinct for the specification and very compact for the body of each active object, which were in fact the ones needing more complex *ad-hoc* structures in the original mapping.

4.4. Space and time efficiency

The advantage gained by the source code factoring strategy outlined above is mostly leveraged in terms of a library of generic templates and a rather simple set of directives for the creation of object instances.

Let us now look at what value this strategy buys us when it comes to space and time efficiency.

To begin with, it is very easy to see that the volume of *source* code that our generator produces for the system is considerably less than that produced by the rules originally proposed in [6]. Figure 1 shows that already with as few as 6 object instances the proposed generator produces some 20% source code less than the traditional rules.

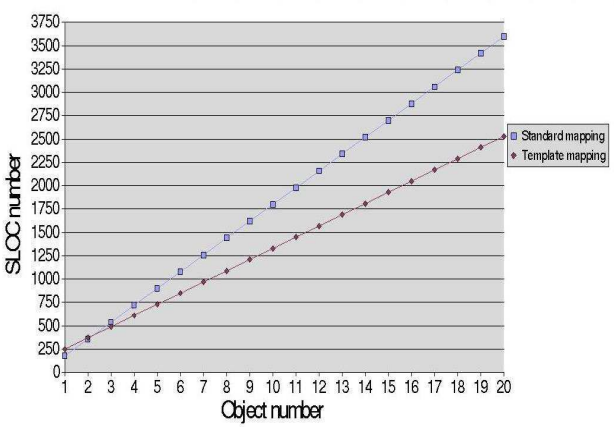

Figure 1. Source code lines generated with the new (template based) generation rules and with the original (standard) ones.

Not surprisingly, the gain is linear in the number of object instances. Where we lose something is in object code size.

The use of hierarchical libraries introduces negligible costs. We have checked on various compilation systems for various targets and have seen that if there is any penalty to be paid, it ranges $80 - 200$ bytes per child unit. A population of 200 object instances (which would look like a large and complex system) might thus incur, in the worst case, a size overhead inferior to 40 KBytes.

Generic instantiation instead may in principle impair both space and time efficiency. Simple experiments have given us sufficient confidence that the execution time penalty to be feared is negligible if at all incurred. The effect on the executable size instead actually depends on the instantiation technique. The Ada compilers we could get access to all use macro expansion as opposed to code sharing. (The reader should note that this choice was not ours to make. We were the users of those compilers, not the designers, and we had no switches to turn options, which, admittedly would have been interesting to have.) In a way, macro expansion causes the executable code size to inflate to a rate that goes inversely linear to the saving obtained on source code factoring. This may cause us to incur an executable size penalty in the lower region of 30% over the size resulting from the traditional generation rules.

Embedded processor technology has entered the age of 32-bit computing about a decade ago and this has luckily changed the perspective with which such a possible size overhead can be looked upon. By way of example, the maximum affordable size for the very costly and low-performance radiation-hardened memory for the previous 16-bit generation seldom exceeded 1 MByte. For such systems, a 30% size overhead might have been very bad news indeed.

Current generation processors for embedded computing in the application domains targeted by HRT-HOOD and thus by HRT-UML afford in excess of 10 MBytes, which undoubtedly lessens the negative impact of a size overhead in the range we might incur, although it does not fully dissipate it.

5. Conclusions

In this paper we have argued that structural compliance to a restricted concurrent computational model especially tailored for the domain of HRT systems can be achieved from model-based specification through design to source code generation.

We contend that this achievement attains progress towards more extensive use of model-based generative programming environments for HRT applications.

We have chosen the Ravenscar Profile as the reference computational model and we have illustrated the impact its restrictions have on the HRT-UML design method.

Other studies have shown and argued that the Ravenscar restrictions avail sufficient expressive power to HRT designers. We agree with this contention and therefore need not defend the choice of the profile ourselves.

The Ravenscar Profile restrictions were originally aimed at achieving source code directly amenable to static analysis and also implementable on highly efficient, robust and certifiable kernels. As the Ravenscar mode of HRT-UML produces Ravenscar compliant code, it goes by itself that we should earn the same bonus in kind.

We evaluated the quality of the generated source code against three dimensions: modularity, factoring and efficiency. In fact, our attaining a good deal of factoring in the source code that maps the realisation of the HRT-UML objects was considerably facilitated by the Ravenscar Profile restrictions, which reduced the width and breadth of the semantic meta-model that underpins the HRT-UML objects.

The evidence reported and the considerations that stem from our critical evaluations give us sufficient confidence that the source code generator that we have devised for HRT-UML has positive value that outweigh the possible drawback.

Overall, we did find a single area of risk in the executable size overhead that generic instantiation by macro expansion might incur. We have indeed noticed that the range of Ada compilers that we could exercise do incur an executable size overhead that we could estimate to some 30% over functionally equivalent code with no generic instantiation.

Looking at what this possible size overhead might mean for the application domains of interest to our method and our source code generator, we have tentatively concluded that the benefits largely exceed the penalty to be paid for them.

References

[1] *Consolidated Ada Reference Manual — International Standard ISO/IEC-8652:1995(E) with Technical Corrigendum 1*, 2000. ISO/IEC 8652:1995.

[2] Ada Conformity Assessment Authority. *International Standard ISO/IEC 8652:1995 - Information Technology - Programming Languages - Ada - Amendment 1 (Draft 8)*, October 2004. `http://www.ada-auth.org/amendment.html`.

[3] A. Burns. Scheduling Hard Real-Time Systems: a Review. *Software Engineering Journal*, 6(3):116–128, 1991.

[4] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Advances in Real-Time Systems*. Prentice-Hall, 1994.

[5] 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 (UK), January 2003. Approved as ISO/IEC JTC1/SC22 TR 42718.

[6] 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.

[7] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000. ISBN: 0201309777.

[8] B. Dobbing and G. Romanski. The Ravenscar profile: Experience report. In *Proceedings of the 9th International Real-Time Ada Workshop*, volume XIX(2), pages 28–32. Ada Letters, 1999.

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

[10] J. Goodenough and L. Sha. 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.

[11] Java Community Process. *JSR-001: Real-Time Specification for Java*, March 2004. `http://www.jcp.org/en/jsr`.

[12] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: a High-Integrity Profile for Real-Time Java. In *Java Grande Conference*, pages 131–140. ACM-ISCOPE, 2002.

[13] S. Mazzini, D. M., M. Di Natale, G. Lipari, and T. Vardanega. Issues in Mapping HRT-HOOD to UML. In 15^{th} *Euromicro Conference on Real-Time Systems*, pages 221–228. IEEE CS, 2003.

[14] Object Management Group (OMG). *UML 2.0 Superstructure Specification*, 2003. `http://www.omg.org/cgi-bin/doc?ptc/2004-01-11`.

[15] OMG. *UML 1.5 Specification*. `http://www.uml.org/#UML1.5`.

[16] OMG. Model Driven Architecture. `http://www.omg.org/mda`, 2004.

[17] B. Sanden. Coping with Java threads. *IEEE Computer*, 37(4):20–27, 2004.

[18] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. Wiley & Sons, 1994. ISBN 0471599174.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Tr. on Computers*, 39(9):1175–1185, 1990.

[20] T. Vardanega and G. Caspersen. Using the Ravenscar Profile for space applications: The OBOSS case. In M. Gonzlez-Harbour, editor, *Proceedings of the International Workshop on Real-Time Ada Issues*, volume XXI, pages 96–104. Ada Letters, 2001.

[21] T. Vardanega, M. Di Natale, S. Mazzini, and M. D'Alessandro. Component-Based Real-Time Design: Mapping HRT-HOOD to UML. In 30^{th} *Euromicro Conference*, pages 6–13. IEEE CS, 2004.

[22] T. Vardanega, J. Zamorano, and J.-A. De la Puente. On the Dynamic Semantics and the Timing Behaviour of Ravenscar Kernels. *Real-Time Systems*, 29(1):59–89, 2005.