A Reinterpretation of Patterns to Increase the Expressive Power of Model-Driven Engineering

Matteo Bordin[1], Marco Panunzio[2], Carlo Santamaria[2], and Tullio Vardanega[2]

[1] AdaCore
46 rue d'Amsterdam, 75009 Paris, France
bordin@adacore.com
[2] University of Padua, Department of Pure and Applied Mathematics
via Trieste 63, 35121 Padova, Italy
{panunzio,tullio.vardanega}@math.unipd.it

Abstract. The model-driven engineering (MDE) paradigm wishes to raise the abstraction level of the user design space, while resting on the automated generation of all lower-level artifacts. Under the MDE approach the focus of verification and validation increasingly verges on models. As a consequence, the expressive power availed to the user is often considerably restricted to ensure that the models are amenable to static analysis. Inherent tension thus arises in the very essence of MDE between the restraints to be placed for a better good on the user-level expressive power and the user need and expectation to be able to operate in a modeling space delivered of platform dependences and constraints. In this paper we contend that a new notion of modeling patterns may help resolve the conflict and increase the expressive power in the user space without jeopardizing the integrity and effectiveness of the transformation process.

1 Introduction and related work

Motivation. Model-Driven Engineering [1] aims to decrease the time and cost of software production and to increase quality, by leveraging on the factorization of best practices in programming and implementation. The MDE paradigm strives to raise the abstraction level of the user space and to generate all lower-level artifacts automatically, source code, analysis models, and documentation alike. MDE wishes to deliver the user from the burden of dealing with platform-specific implementation details, and to concentrate instead on the (platform-independent) specification of the solution. The assumption behind this vision is that the implementation may be largely if not completely delegated to the automation capabilities of platform-specific development frameworks.

At present however, the adoption of MDE for the high-integrity application domain is still a challenge. Already in the general case in fact, it may be overly difficult to provide sufficient assurance that all properties attached to the user model and validated at that level of abstraction be correctly propagated throughout model transformations, and preserved upon deployment and execution. This provision demands a level of control over and proof of the production process, which is difficult to attain for the general user.

State of the art. The most prominent efforts in modeling languages for real-time systems in the industrial landscape to date are AADL and MARTE. AADL [2] focuses on the modeling of schedulable entities, of information and control flows, and on the analysis thereof. AADL has also recently been augmented with specific annexes targeting behavioral aspects and error treatment mechanisms. AADL conveys all user concerns, such as scheduling, flow and behavioral modeling, into a *single* modeling view. The main advantage of this choice is the comparative simplicity and cohesiveness of the modeling language: the AADL syntax is particularly compact, and each semantic concept can easily and directly be expressed with a single combination of syntactic constructs. The single-view modeling of multiple concerns however limits the power of abstraction considerably and pushes it down to the implementation level. The abstraction level of AADL models thus gets downcast to that of the underlying implementation as intended by the target execution platform and the accompanying theories of analysis, which is quite contrary to the intention of MDE.

MARTE [3] is an OMG effort to bridge schedulability-oriented modeling with system-level aspects such as flow analysis and software/hardware interaction. (MARTE can in fact be used in conjunction with SysML [4].) As of September 2008 the MARTE specifications are in official beta status. MARTE suffers from the gigantism typical of several OMG standards. As in UML, a MARTE model is comprised of several views, the consistency of which is *not* assured by the underlying metamodel. Moreover, even if vastly more expressive than AADL (especially for time-related semantics), numerous syntactic constructs in MARTE insist on one and the same semantic concept and thus overload it. These characteristics make MARTE models rather complex to understand. Ultimately, the semantics expressible with MARTE is close to that assumed in common scheduling analysis theories.

While both AADL and MARTE provide platform-independent ways of modeling software systems in manners amenable to static analysis, the abstraction level of their modeling space is restrained by constraints arising from the execution semantics intended for the target platform. In fact, in both AADL and MARTE the abstraction level at PIM is almost equivalent to that at PSM. That closeness eases the preservation of model attributes and properties across model transformations, but at the cost of permitting only a shallow distance between PIM and PSM.

The last modeling language we consider is HRT-UML/RCM [5,6], the authors' own proposal, an MDE infrastructure devised in the ASSERT project [7] to defy this challenge especially. In ASSERT, HRT-UML/RCM was used for the development of an industrial-scale real-time embedded system by one of the major prime contractors in European space industry.

HRT-UML/RCM aims to: (i) provide a design environment in which the user solely operates in the PIM space, with the only exception of the specification of hardware configuration and application deployment; and (ii) support an MDE methodology characterized by principles of correctness by construction [8] and of property preservation [9], across all model transformations including at run time.

The HRT-UML/RCM model space does not allow any semantic variation points: the run-time semantics expressed in its models thus always is completely defined.

In HRT-UML/RCM, the user specification of the PIM is declarative, while the transformation process applied to it corresponds to an implementation designed to be provably correct by construction. The resulting product consequently does not need to be verified a posteriori on a per-system basis, but only requires a single per-platform validation, with important cost savings for the developer.

HRT-UML/RCM has for now elected to produce a single PSM from the PIM. space, though other PSM may in principle be generated, for instance to address dependability, safety and security concerns. In HRT-UML/RCM the PSM is also used as a *Schedulability Analysis Model* (SAM). The SAM represents the semantics of the system model to the extent of allowing static analysis of the feasibility and sensitivity of its timing behavior. The SAM generated in HRT-UML/RCM is comprised of a set of comparatively simple building blocks, which, through correct-by-construction composition, may arrive at encompassing arbitrarily complex execution semantics.

HRT-UML/RCM seamlessly integrates round-trip support for feasibility and sensitivity analysis and makes it start and end at the PIM [10] (see figure 1). While the analysis is of course made on the SAM, its results are propagated back to the PIM, which is possible as the entire model transformation logic is deterministic and reversible, hence it may be easily followed backwards.

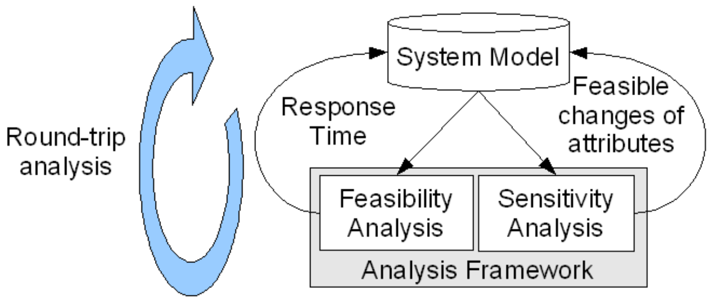

Fig. 1. Round-trip timing analysis in HRT-UML/RCM

HRT-UML/RCM also supports automated generation of source code. This leg of the transformation process starts from the SAM, which eases the provision of constructive proofs that the system at run time does correspond to what was analyzed and deemed feasible in the SAM. In the case in instance the complexity of the code generator is modest since the SAM is very close to the system at run time and the code generation engine makes extensive use of simple patterns [11].

A factor that greatly facilitated our attainment of correct-by-construction transformations was the decision to hoist the observance of the RCM constraints from the PSM space up to the PIM. This decision however has the downside that it pushes back onto the user the need to think in implementation terms (so that the RCM restrictions are not violated) in contrast with the promise of delivery from those very concerns.

Contribution. The challenge we wish to defy is to raise the user space to a higher level of abstraction (closer to the problem domain and more distant from the constraints of implementation) while guaranteeing preservation and assurance of properties across deeper model transformations. In this paper we discuss the role that *MDE patterns* may play to this end.

Previous work on the use of patterns in real-time systems [12,13] predominantly if not exclusively considered *design patterns* in the way they were promoted by the "Gang of Four" [14]. We contend that such a view fails to take full advantage of the emerging MDE paradigm. We show additional classes of potentially useful patterns, and attempt an initial classification of them from the broader perspective of the transformation space.

The remainder of the paper is structured as follows: in section 2 we define what we mean by "expressive power" in the context of MDE; in section 3 we draw a tentative classification of MDE patterns against the hierarchy of transformations implied in the process; in section 4 we discuss a few example patterns in some details and finally we draw some conclusions.

2 Expressive power in Model-Driven Engineering

We consider the expressive power of a language to relate to its economy of expressions: the more synthetic the language entities and the denser their semantic contents, the greater the expressive power. By this definition, the keywords of a programming language are much more expressive than the words in the instruction set of the target processor. In the context of programming languages the transformation of a more expressive program text into a lesser one is taken care of by the compiler. It is a well-know observations that the very existence of a compiler for a given target permits to implement other programming languages equipped with greater expressive power [15].

The expressive power thus is the capability of expressing, synthetically, high-level concepts with a finite set of language terms with known meaning and with the guarantee that they can be correctly translated into a semantically equivalent set of entities that belong to an underpinning implementation language.

This very principle lies at the heart of the MDE paradigm too. The PIM space may exploit a dictionary of technology- and implementation-independent terms, which model transformation translates to terms that belong to a specific execution platform. Source code is only one of many possible PSM produced by a model transformation of a PIM. In general it is possible to generate a set of PSM each of which serves a different purpose. Each PSM may thus represent the implementation of the PIM at a distinct level of abstraction, or viewpoint, each of which focusing only on the concerns of interest to selected view-specific stakeholder.

The expressive power of PSM is often constrained by the level of formalism required to permit the sound application of domain-specific analysis techniques. For example, constructs that incur non-determinism or unbounded execution behavior may be removed from the PSM language in the prevailing interest of facilitating static analysis.

In general, two opposite approaches can be pursued to increase the expressive power availed to the PIM space:

- granting the maximum possible (in principle, full) freedom of expression to the user and then verifying *a-posteriori* whether the user model can be successfully turned into a semantically-equivalent and legal PSM;
- capturing all of the constraints that propagate up from the PSM and striving to relax all of them for which a transformation pattern may be devised which may be proven correct *a-priori* and whose eventual overhead may be deemed acceptable to the application.

In the former approach the MDE infrastructure provides the designer with expressive power as large as the user can have. In this case however there is no guarantee that a legal PSM may be obtained from automated transformation of the user model. The verification must be therefore made a posteriori and on a per-model basis.

With the latter approach instead, the infrastructure clearly continues to restraint the expressive power, but for the benefit of a-priori guarantees that any user model in the PIM scope can be automatically transformed into a legal PSM. The metamodel is then the key element to ensure that the user model space is exclusively populated with legal entities, attributes and relations.

In HRT-UML/RCM the PIM modeling space is directly restrained by the bottom-up propagation of semantic constraints from the underlying computational model, cast onto the RCM metamodel. The RCM originates in language-neutral terms from the Ravenscar Profile of the Ada programming language [16]. The Ravenscar profile: (i) forbids the use of language constructs that may incur non-determism or unbounded execution time; (ii) only allows asynchronous one-way communications mediated by shared resources equipped with a deterministic synchronization protocols, like the Priority Ceiling Protocol (direct inter-task communication are thus prohibited). RCM further requires threads to have a single suspension and a single (cyclic or sporadic) source of activation events. All those restrictions are imposed on the PSM to ensure that all models in that space be statically analyzable. We want to be able to turn every possible user model in the PIM space into a semantically equivalent PSM, which also be correct by construction against the RCM restrictions.

When the applicable RCM constraints propagate up to directly restrict the user modeling space, the expressive power of the PIM can only be marginally larger than a simple bottom-up projection of the expressive power of the PSM (cf. figure 2.a).

HRT-UML/RCM offers a set of declarative stereotypes to increase the abstraction level in the PIM space. For example, provided services are decorated with attributes that express their intended concurrent behavior and timing properties without the user having to bother with how to implement them. Nonetheless, several restrictions of the RCM (which thus pertain to the PSM) still directly apply to the PIM. For example, the RCM forbids the creation of deferred operations with out parameters (i.e., operations executed by a server-side thread which may return values to the caller) for that would incur synchronous blocking semantics in violation of the Ravenscar restrictions.

We believe that a more advanced use of MDE patterns may help us extend the expressive power of the PIM. We want to attain the maximum possible increase while maintaining the guarantee that *all* the user models expressible in the PIM space can be represented as (arbitrarily complex and yet correct by construction) compositions of legal PSM entities. A deterministic yet efficient function must exist to transform the

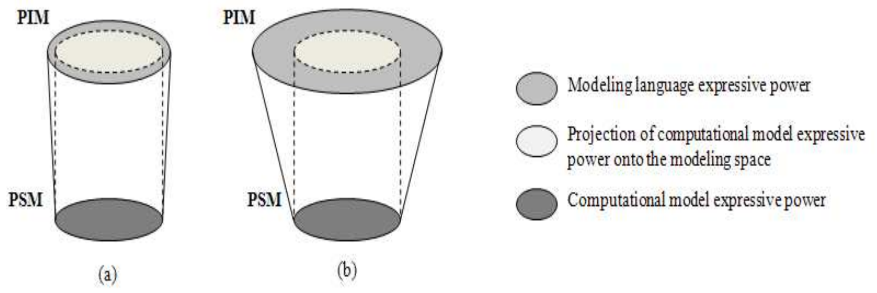
PIM
PSM
(a)
PIM
PSM
(b)
Modeling language expressive power
Projection of computational model expressive power onto the modeling space
Computational model expressive power

Fig. 2. Relation between expressive power of computational model and modeling language. Figure (a) represents a modeling space which is a projection of the underlying computational model: this is the current situation of HRT-UML/RCM. In figure (b), we represent our goal: the user-level modeling semantics is wider than the projection of the PSM computational model; but it is still possible to trace such semantics to an equivalent representation that respects the RCM constraints.

constructs that belong in the extended PIM space into a combination (thus, intuitively, a composition pattern) of those primitively present in the PSM (see figure 2.b).

Granted, the additional expressive power can only come at some cost: the larger the distance between the declarative language of the PIM and the implementation language of the PSM, the greater the time and space overhead of the model transformation, of its code products and of and its verification effort.

3 Classification of patterns

Let us first introduce a tentative classification of patterns in the MDE landscape. With this classification we maintain that MDE patterns distinguish themselves by the abstraction level at which they are applied. The abstraction level at which any given pattern is considered to belong thus becomes the central element to decide their goodness of fit to serve our objective.

In our current classification, we recognize:

- *determined patterns*: this kind of patterns are explicitly instantiated by the user, who recognizes their possible use in the current design and manually embeds them in the model as a solution to the problem at hand
- *executive patterns*: this kind of patterns are the result of the identification of features or constraints specific to the platform(s) of interest. They are directly included in the metamodel and the associated transformations, for they provide satisfactory solutions to known implementation-specific problems. These patterns are not directly available to the user, who is not aware of their use
- *declarative patterns*: this kind of patterns are solutions to recurrent problems in the application domain. The design infrastructure is aware of these patterns, which are offered to the user as built-in components that can be safely embedded in transformations of the user model.

Determined patterns represent user-perceived solutions to recurrent problems in the modeling space of the application. The user recognizes a specific problem in the application requirements and manually augments or adapts the model to host an instantiation of the desired pattern(s). The design patterns discussed in [14] clearly fall in this category. It is important to notice that, following our definition, this kind of patterns do not increase the expressive power of the MDE infrastructure, since they are user-level constructions that result from the assembly of primitive entities. Consequently, only the latter actually “exist” in the model. The user-level constructions instead have no direct representation across model transformation since the MDE infrastructure is unaware of their existence.

Executive patterns encode traits of the implementation domain of interest. A computational model, a set of constraints on thread activations and suspensions, archetypes to factorize common behaviors, are all example of executive patterns. Executive patterns distinguish themselves because they must necessarily be encoded in the metamodel and fixed once and for all upon its creation. Since they are part of the metamodel, which constrains the legal design space, they determine the PSM expressive power. As the MDE paradigm wishes to lift the abstraction level of the user space away from implementation details, the executive patterns need not be directly visible to the user.

Declarative patterns are themselves solutions to a recurrent problem of the application domain. More specifically, the designer recognizes a known problem in the system specification and uses one of these patterns to solve it. Declared patterns must be semantically understood by the user and recognized as solutions to specific problems in the application space. They however require no implementation from the user, who just need to cause them into existence. It is the process of model transformation in fact that instantiates the required patterns and embeds them in the user model. For this reason declarative patterns must pose no obstacle for the products of model transformation to stay within a legal, correct by construction, design space. The most effective way to achieve this is for declarative patterns to exist in the metamodel and consequently be used as constructive elements of the model transformation logic. Following our definition, declarative patterns augment the expressive power availed to the user, since they result from constructs (e.g.: attributes) that model transformation resolves in a predefined and correct by construction composition of primitive entities of the PSM. Furthermore, the semantics of a declarative pattern is known and well defined and the enforcement of legal conditions for its application can be assured by the MDE infrastructure itself; both these characteristics are instead contemplated in the application of determined patterns.

One of the goals of this categorization is to let the reader appreciate how important those patterns may be to the MDE process and how much they may influence the expressive power availed to the user.

As we have seen, executive patterns basically determine the expressive power available at the PSM level, that is the (low-level) abstraction layer of the entities that populate the system at run time.

Determined patterns live entirely in the user-level design space; they are composed of primitive entities that can legally populate the PIM and therefore have scarce importance in the MDE process, since they do not augment the expressive power.

Declarative patterns instead arguably are the most important category of patterns in the context of an MDE process: they provide the user with a powerful abstraction that is automatically instantiated by model transformations and therefore fit perfectly into the model(s) product of the transformation.

It is then clear that declarative patterns are a crucial instrument for us to increase the expressive power we may avail to the user. Expressed graphically with reference to figure 2, we are increasing the slope of the lines that join the PSM space to the PIM.

One additional benefit of this classification is that makes it clear where the instantiation of each pattern occurs. Figure 3 provides a diagrammatic representation of this element of information: determined patterns clearly populate the *application layer*, since they are composed of legal entities of the user-level design space; executive patterns map or encode features of the execution environment, whether software (kernel or middleware or both) or hardware; declarative patterns, finally, reside at PIM level, but outside the direct projection of the PSM expressive power and thus require model transformation to come in existence as a correct by construction assembly of legal PSM entities and constructs.

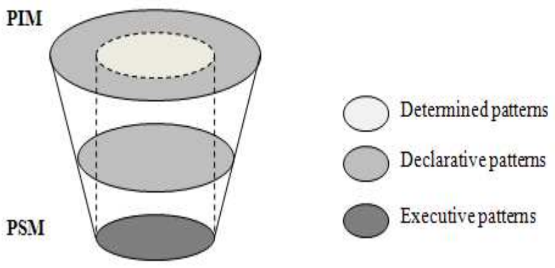

Fig. 3. Patterns in model-driven engineering. Determined and declarative patterns are avaliable in the user modeling space. Determined patterns result from direct projections of the PSM expressive power. Declarative patterns exist beyond the projection of the PSM expressive power and require model transformation to be implemented in terms of legal PSM entities. Executive patterns solely exist in the PSM level.

Let us now determine whether and to what extent the user should be aware of patterns, according to their class of belonging.

Determined patterns are obviously known to the designer, who is the primary and sole actor in their use. Conversely, executive patterns are intentionally hidden to the user. The situation is not that clear instead for declarative patterns. Should the designer be aware that the decoration of some model attributes triggers the activation of a declarative pattern? Additionally, should the designer explicitly require the activation of a pattern or else simply rely on the intelligent support of the framework to recognize the conditions for its activation? Two analogies may help us illustrate the differences between these two approaches.

One of the classical optimization performed by a compiler is to move the invariant part of a loop outside the loop itself. The average programmer need not be aware that this optimization is performed. Conversely, let's imagine a programming language that prescribes that remote operations be flagged with a *remote* keyword; in this case the programmer intentionally determines the generation of stubs, skeletons, and all other elements required to perform a remote invocation. At present we do not have a strong position on the *entire class* of declarative patterns, and we are inclined to believe that the issue should be evaluated on a case by case (i.e., per pattern) basis.

4 Pattern catalogue

This section briefly discusses examples of patterns from the categorization we have just proposed. The examples we have chosen reflect problems typical of the real-time application domain. We concentrate on declarative and executive patterns, because determined patterns should be well known to the reader.

4.1 Partitions and communication filters

While the notions of partition and communication filters are not new to software engineering, especially in the high-integrity domain (see for example the ARINC-653 standard [17]), they still have to find their place in a model-based development.

The use of logical and physical partitions permit to attribute software and hardware components to distinct levels of criticality so as to guarantee the required level of isolation in time, space and communication among them.

The notion of partition must be part of the metamodel itself, because the designer must be able to consciously allocate multiple executable entities within given partitions. In that manner, the entities included in a partition inherit the partition criticality and benefit from the partition-level isolation mechanisms. Neither the causing of criticality inheritance nor the modeling of isolation mechanisms, however, are to be explictly performed by the user: they can instead be easily realized by way of model transformation. The notion of *partition* does thus reflect a declarative pattern.

Communication filters are tightly related to partitions. When a lower-criticality partition establishes a communication link with a higher-criticality partition, the integrity of the exchanged messages should be subject to verification. The execution of the higher-criticality partition may in fact be affected by the computation required by a lower-criticality partition. To only permit allowable communications (operation requests or reports of results), filters are suitably interposed between communicating partitions to perform all of the necessary verification on the permissibility of the communication.

Filters are part of the metamodel. However they are not to be used explicitly by the user, who need not even be aware of their existence, but rather they are put into place by the tail end of the model transformation process. The notion of *filter* does thus reflect an executive pattern.

4.2 Callback

The purpose of the *Callback* pattern is to extend the PIM expressive power beyond the projection of the PSM modeling space. The pattern can be categorized as both an executive and declarative pattern. It does not require any specific action from the designer (as thus equates to an executive pattern), but its application has consequences which must be known to the user, as they impact the software architecture and the interpretation of analysis results.

Let us use a simple example to illustrate the Callback pattern, which we base on the classical producer-consumer archetype. Both the Producer and the Consumer have a single method, respectively `produce` and `consume` (cf. figure 4). The operation of `produce` consists in (a) producing an item, (b) passing it to the consumer and (c) adapting its own behavior according to the Consumer's feedback. The operation of `consume` consumes the item and returns its feedback via an `out` parameter, `Feedback`.

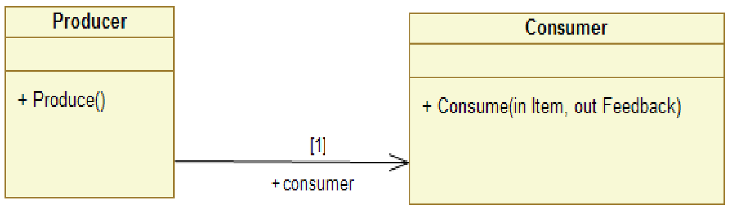

Fig. 4. Callback pattern: class diagram prior to the application of the pattern.

Following the HRT-UML/RCM modeling semantics for components, we declare the concurrent semantics on ports of provided services: the port providing `produce` is marked ≪cyclic≫, meaning that a dedicated task with a constant periodic release calls `produce`. The port providing `consume` is instead marked ≪sporadic≫, meaning that its invocation by a caller causes a sporadic task to be released to actually execute `consume` (cf. figure 5). Additional non-functional concerns, such as tasks priority and period or minimum inter-arrival time, are addressed by decorating the provided port of each component with specific attributes (which we omit in this discussion).

It is worth noticing at this point that the semantics expressed in this user model is *not* Ravenscar-compliant. The reason is that operation `consume` requires a synchronous communication (since its profile includes an `out` parameter), but the corresponding port is marked ≪sporadic≫, which makes it deferred in HRT-UML/RCM and thus necessarily asynchronous. The semantics intended by the user model is that of a synchronous deferred communication, which is forbidden in RCM. Interestingly however, a simple stage of model validation can notice the problem and trigger appropriate actions to resolve it.

To solve the problem and thus satisfy the user without incurring violations of the RCM restrictions, we first perform an *automated* model transformation on the class

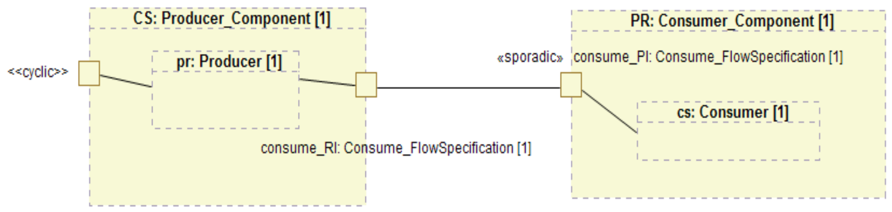
CS: Producer_Component [1]
pr: Producer [1]
<<cyclic>>
consume_RI: Consume_FlowSpecification [1]
«sporadic»
PR: Consumer_Component [1]
consume_PI: Consume_FlowSpecification [1]
cs: Consumer [1]

Fig. 5. Callback pattern: system model prior to the application of the pattern.

diagram (which is part of the so-called functional model in HRT-UML/RCM). The transformation changes the profile of the `consume` to accept a callback (i.e., a function pointer) instead of the `Feedback` out parameter. Its action semantics is then changed to: (a) declare a local variable in place of the missing `out` parameter; and (b) invoke the callback at the end of its execution, passing the above variable as its in parameter.

The action semantics of `produce` is then changed accordingly: in correspondence to the invocation to `produce` it is split into two separate methods. The action semantics coming *after* the invocation to `consume` is enclosed into `consume_callback`, which is the callback passed to `consume` (cf. figure 6).

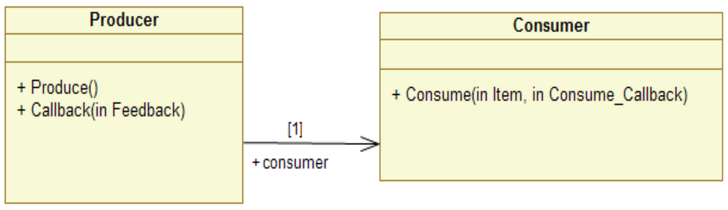

Fig. 6. Callback pattern: class diagram after application of the pattern.

The concurrent semantics declared in the component ports remains the same for those that provide `consume` and `produce`. The port providing `consume_callback` is instead marked ≪sporadic≫ so that a dedicated task may ensure a prompt response to the invocation of the callback from `consume`.

The resulting model is Ravenscar-compliant again as the services provided by the ports marked as ≪sporadic≫ do not include `out` parameters (cf. figure 7) anymore.

At this point an additional model transformation may generate the SAM and assign the user-specific functional behavior to it. Dedicated tasks are created for `produce` (cyclic), `consume_callback` (sporadic) and `consume` (sporadic) and the respective methods are allocated to the fully-legal main operation of the respective tasks. Message queues protected against concurrent access are created for `consume` and `consume_callback` as the means to implement asynchronous communication between the caller and the sporadic task on the side of the callee. A further shared re-

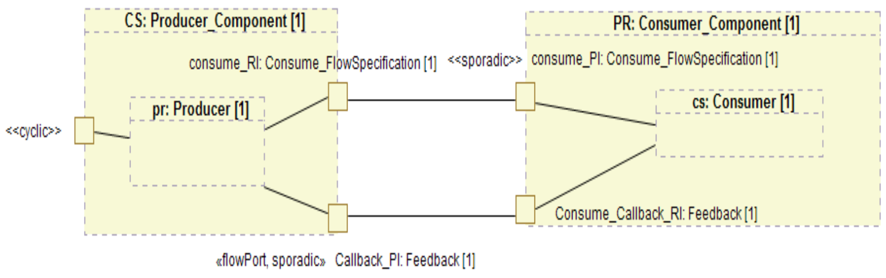
CS: Producer_Component [1]
consume_RI: Consume_FlowSpecification [1]
<<sporadic>>
PR: Consumer_Component [1]
consume_PI: Consume_FlowSpecification [1]
cs: Consumer [1]
<<cyclic>>
pr: Producer [1]
Consume_Callback_RI: Feedback [1]
«flowPort, sporadic» Callback_PI: Feedback [1]

Fig. 7. Callback pattern: system model after application of the pattern.

source may be automatically generated to safeguard the concurrent access the tasks behind `produce` and `consume_callback` to application data that the split of the single user-level operation should not duplicate.

Source code is finally generated from the transformed model and from the SAM; no code is instead generated from the original user model.

5 Discussion

By the introduction of the executive and declarative pattern categories, we achieve two results which are beyond the reach of classical determined patterns.

Executive patterns, like the Filter pattern, relieve the designer the need to specify complex yet recurrent parts of the software behavior, and rather introduce them directly in the PSM (and, possibly in the source code too).

Declarative patterns instead, serve two distinct purposes. First of all, they reduce the amount of modeling effort required for the designer to express the semantics needed to solve an application-level problem: this is for example the case of the Partition pattern.

Declarative patterns may also help extend the expressive power availed at PIM well beyond the perimeter permissible to the PSM: this is the case of the Callback pattern.

In fact, we see declarative patterns as a most promising pattern category for high-integrity systems in general and real-time systems in particular. They have the potential to release the user-level modeling process from (some of) the restraints that are propagated upwards from the underlying analysis theories, and thus extend the expressive power availed to the user. In practice the restraints that may be lifted are those for which declarative patterns exist which permit model transformations that provably preserve the properties of interest down to the final implementation.

Interestingly enough, declarative and executive patterns are both realized by means of property preserving model transformations and take the form of correct by construction aggregates of primities entities of the metamodel. While those patterns are not primitive entities themselves, they do exist in the metamodel space because it is their very existence under the semantic constraints enforced by the metamodel that asserts their legality. A key implication of this stipulation is that the compositional logic of the model transformations must be defined as an integral element of the metamodel itself.

What we currently see as the main limitation to the full application of declarative patterns is the impact they may have on the functional (and not only architectural) specification of the system. We have seen a glimpse of this problem in the application of the Callback pattern, which caused us to break a single functional operation into two distinct parts.

To date, mainstream modeling technologies have failed to provide a full and usable representation of action semantics in the metamodel space. This limitation constitutes a technological (though not conceptual) hurdle to our endeavor.

In several, perhaps most cases, declarative patterns cannot be silently applied by the modeling infrastructure as they may considerably increase the distance between the user space and the part of the PSM that is the product of automated model transformation. The larger the distance the more complex to understand the end results of the transformation, in both qualitative and quantitative terms. For example, the application of the Callback pattern not only modifies the functional specification provided by the user, but also impacts the concurrent and synchronization properties of the system by creating an additional task and an additional shared resource. The modeling infrastructure should thus justify all model transformations and provide evidence of traceability between levels of abstraction.

6 Conclusions

The production of formal analysis models is a complex task which requires to verify that (i) the designed model does not evade the boundaries of the semantics permitted by the underlying analysis theories; and (ii) the semantics is preserved at each abstraction level crossed by transformation.

The preferred way to satisfy both requirements is to constraint the modeling space by directly projecting the expressive power of the PSM onto the PIM modeling space (with the side benefit of easing the automated production of the PSM).

We introduced two new categories of patterns specific of the MDE context to add to the well-know category of determined patterns that stem from [14]. Executive and declarative patterns are meant to address other issues than those targeted by classical design patterns.

Executive patterns are expected to deliver the user from the need to specify parts of the architecture and functional behavior of the application by embedding consolidated solutions directly in the PSM (possibly the source code).

Declarative patterns instead extend the expressive power of the PIM by relaxing semantic constraints that hold on the PSM. The implementation of the semantics implied by declarative patterns is naturally realized by automated model transformation.

We discussed three specific instances of patterns in the above categories, which address problems recurrent the real-time systems domain.

The introduction of those new categories of patterns promises to increase the expressive power attainable at user level and an interesting new dimension of automation in the model-driven engineering of real-time systems. The full exploitation of those patterns requires a highly integrated modeling infrastructure, which includes a modeling language, and a set of proven and correct by construction model-to-model and model-to-code transformations.

References

1. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2) (2006) 25–31
2. SAE: Architecture Analysis and Design Language. `http://la.sei.cmu.edu/aadl/currentsite/aadlstd.html`.
3. OMG: UML profile for MARTE. (2007) `http://www.omg.org/cgi-bin/doc?ptc/2007-08-04`.
4. OMG: SysML specification. (2007) `http://www.omg.org/cgi-bin/doc?formal/2007-09-01`.
5. Bordin, M., Vardanega, T.: Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In: Reliable Software Technologies - Ada-Europe. (2007)
6. Panunzio, M., Vardanega, T.: A Metamodel-driven Process Featuring Advanced Model-based Timing Analysis. In: Reliable Software Technologies - Ada-Europe. (2007)
7. ASSERT: `www.assert-project.net` (2004-7)
8. Chapman, R.: Correctness by Construction: a Manifesto for High Integrity Software. In: ACM International Conference Proceeding Series; Vol. 162. (2006)
9. Vardanega, T.: A Property-Preserving Reuse-Geared Approach to Model-Driven Development. In: 12th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications. (2006)
10. Bordin, M., Panunzio, M., Vardanega, T.: Fitting Schedulability Analysis Theory into Model-Driven Engineering. In: Proc. of the 20th Euromicro Conference on Real-Time Systems. (2008)
11. Pulido, J.A., de la Puente, J.A., Hugues, J., Bordin, M., Vardanega, T.: Ada 2005 Code Patterns for Metamodel-based Code Generation. Ada Letters **XXVII**(2) (2007)
12. Zalewski, J.: Real-time Software Design Patterns. In: 9th Conference on Real-Time Systems. (2002)
13. Sanz, R., Zalewski, J.: Pattern-based Control Systems Engineering. IEEE Control Systems Magazine **23**(3) (2003) 43–60
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison - Wesley (1995)
15. Pratt, T.W.: Programming Languages. Design and Implementation (Second Edition). Prentice-Hall (1984)
16. Burns, A., Dobbing, B., Vardanega, T.: Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. Technical Report YCS-2003-348, University of York (2003)
17. ARINC: Avionics Application Software Standard Interface: ARINC Specification 653-1 (2003)