

On the Goodness of Fit for High-Integrity Systems: Reflections on the Design and Evolution of Model-Driven Tools

Sue Maurizio¹ and Tullio Vardanega²

¹ Soluta.Net (smaurizio@soluta.net)

² Università degli Studi di Padova (tullio.vardanega@math.unipd.it)

Abstract. The reflections presented in this paper stem from an integrated industrial and academic R&D project that aimed at the development and use of model-based support for the design of high-integrity applications. We discuss some observations on and lessons learned from the difficulties encountered in the project in the pursuit of that goal. We focus in particular on the evolution of the tool infrastructure, the learning curve for users, the optimization of the transformation results.

1 Introduction

This paper aims to reflect on lessons learned from the authors' experience in the design and use of model-driven methodologies and tools for use with high-integrity applications. One of the main products of the ASSERT project [Pro], an Eclipse-based model-driven tool environment, was indeed subject to a gratifyingly intensive evaluation by the industrial members of the project. The ASSERT (Automated proof-based System and Software Engineering for Real-Time systems) project was partially funded by the European Commission in the context of the 6th Framework Program. The tool was built following the Model-Driven Architecture (MDA) methodology, promoted by OMG [OMGa].

This paper addresses two important questions, which stand at the very core of the MDA movement:

- *Why should I use MDA?* When a user develops an application using a model-based tool, she wants it to be quick to implement and yet to be optimized, efficient and effective. An automated transformation can hardly have the intelligence of an expert developer, it is not capable of exploiting the stratagems learned over years of professional work, it is not capable of devising complex ad-hoc optimizations.
- *What should I expect from the use of MDA?* The learning curve of a modeling tool is not so far from a beginner's approach to a modern programming language like Java. The complexity of MDA tools however tends to grow as the user demands and expectations also grow: this phenomenon makes the technology progressively more difficult to use and the practice and discipline required for the use of the tool progressively more difficult to learn. Often,

the general feeling of the users is one of deception: the initial goal (and result) of the tool seemed to be making life simpler but, once you learn how to use the various diagrams and languages that compose it, along with the mass of functionalities and rules that go with it, life seems to get considerably more complex than before.

When considering the weight of the arguments that surge from the newbies of the MDA paradigm, but which can likewise be formulated by professionals who followed the birth and growth of model-driven tools and know their evolution path and limitations, defending this methodology may become a challenge.

This paper is not meant to provide a guideline for the development of model-based software, but rather a collection of observations and ideas to make this task a little easier and, possibly, successful.

2 Coping with Increasing User Requests and Product Customization

A good way to reflect upon the goodness of fit of tools for model-driven development is to observe their evolution in time. An interesting example is represented by the tool realized for the ASSERT project: the project required a development environment suited for high-integrity real-time systems, based on a set of formally proven rules to apply in the model transformation process and on the support for model-based static analysis and round-trip engineering (see [Bor06] and [BPV08] for details).

The tool promotes a high-level development of the system and, augmenting the MDA guidelines, produces a formal, provable and analyzable, representation for a specific platform; in our case, this platform is represented by the Ravenscar Computational Model (RCM), a set of rules and constraints derived from the specification of an Ada tasking profile and then generalized for application to any programming language ([BDV03]). To sum the RCM up in brief, the models generated by the framework obey a set of principles that make it fit for a static analysis, thereby reducing the extent of verification to be performed by test.

At the beginning of January 2007, the modeling tool realized for the ASSERT project provided the base functionalities for specifying the structural features of the system, namely its composition in classes and containers at various levels of abstraction, and the capabilities for attaching non-functional attributes to containers and thus determine the allocation of threads of control, the setting of logical and physical partitions and the directives for deployment.

As a matter of fact, though, the expressive power provided to the end user at that point in time covered nothing more than the simple projection of the RCM on the user modeling space; as Figure 1 shows, new requirements by the industrial partners leaned us toward a broadening of the user space.

Some of the needs voiced by the industrial partners in the project concerned limitations of the tool itself (mostly due to its inevitable lack of maturity); most of the other observations however regarded the wish to circumvent the need for

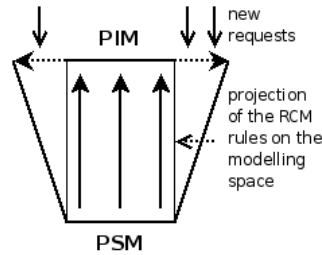


Fig. 1. New requirements forced the modeling space to be expanded beyond the limits imposed by the RCM.

the model space under user control to strictly comply with the RCM, which were felt to fastidiously limit the expressive power of the designer.

It is of course normal during the evolution of a modeling tool to receive requests for improvement from actual and prospective users. Those requests however may pose a serious problem to the tool designers. How can those requests be satisfied without breaking the integrity of the very concept of the tool infrastructure?

A possible solution comes from getting inspiration from one of the most common programming practices: it is sufficient to create a set of libraries, possibly built using the base functionalities provided by the tool, containing architectural solutions to a range of known and recurrent design problems. If the tool is sufficiently mature, the response to this demand can be constructed using its initial expressive power (and thus without altering the initial metamodels or the code generation engine). An important inherent risk in this approach is that the generated code may be inefficient due to the unfit nature of the original model transformations; the user can experience considerable and unacceptable performance differences (in sizing, timing and traceability) between an automatically-generated application and one developed by hand by a domain expert, who knows the peculiarities in the target domain inside out and knows how to solve them effectively.

A different solution can thus be chosen to solve the problem satisfactorily: to integrate the desired user-level functionalities *directly* in the tool as native (user-level) features. This choice requires one to augment the expressive power available to the user, but also to increase the complexity of use. This choice also has some impact on the model transformations and, frequently, on the supporting metamodels. A distinct consequence of this approach is that we may have to resort to a number of singular, specific functionalities (yet all individually crucial to the implementation of some user-level functionality). This event often means that a particular functionality, introduced to permit the salvation of a single (high-level) application need, incurs four serious problems:

1. it contributes to the overall complexity of the modeling environment many times its modest magnitude;

2. it will extend the time needed to learn how to use the tool augmented with the additional functionalities;
3. it will increase the execution time of the model transformations.

An example to illustrate this concept directly comes from the ASSERT experience: one of the goals that were planned for the last year of the project lifetime consisted in the realization of support for the automated (programmed) scheduling of actions: the initial requirement was to allow the execution of some actions to be explicitly scheduled at predefined time intervals, independently of the state of execution of the rest of the system. More precisely, the user intent ultimately consisted in realizing a system conceptually composed of three logical entities, each one dedicated to a specific role in a navigation, guidance and control module of a given device in an aerospace application; each of the three entities was assigned a distinct execution rate and a communication flow with one or more of the others. In Figure 2, each band in the time-line graph represents one entity, while the circles represent the activation instants for the three components and the arrows illustrate the periodic data exchanges among them. For example, the control entity performs some computations every 10 ms and it receives some data from the navigation and guidance components every 100 and 1000 ms respectively.

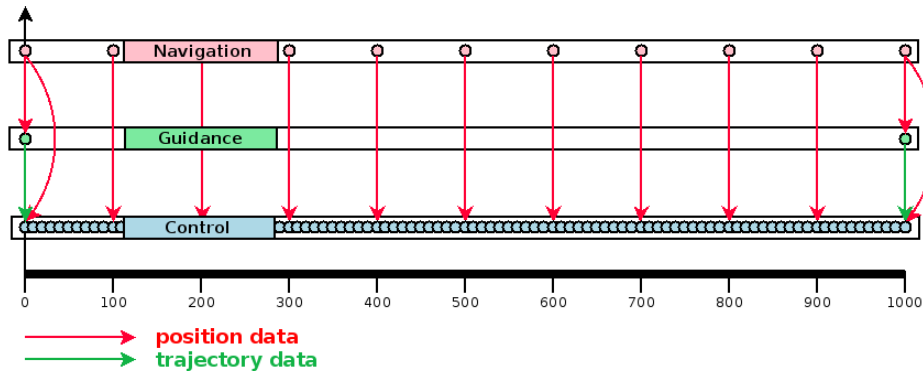


Fig. 2. The behavior of the navigation, guidance and control components.

The requirement wanted us to provide native (user-level) means for the execution of the activities to be performed by the three entities to be scheduled upon the arrival of a periodic event. The user did not want tasks to be used to carry out the execution. At the time this requirement was expressed, the framework that was developed for the ASSERT project already provided some *architectural patterns* for the modeling of control flows and shared resources, normally employed by the control flows themselves to realize asynchronous communications. Various technical notes written and exchanged at the time illustrated different ways to meet the user requirement through building blocks that were already provided

by the framework, showing that there was very little advantage in introducing specific ad-hoc support for the additional functionality.

We experienced the opposite situation, instead, for another additional feature required by the industrial partners in the project: the run-time monitoring of the execution time of threads of control. In this case, the framework already offered a model-driven procedure for static analysis of the execution time [Pan06], but no functionalities were supported to protect the system in case the stipulations made at analysis level were violated at run time. The feature we introduced to meet this requirement provides a way to formally stipulate how the system should react when violations of stipulations would occur at run time; as a consequence, the set of functionalities offered by the tool were significantly augmented, which made it more complete.

These two examples made us reflect on the essential quality of the modifications that can be done to a modeling tool and on the importance of thoroughly studying the way in which the solutions can actually come to fruition: while, in the latter case, native support to the functionality appeared to be fully justified and also confirmed a posteriori, in the former case we could have solved the problem suggesting and documenting a way to implement the desired feature using already-existing functionalities, thus harvesting important advantages:

- the transformation logic from the input model to the generated code would have been far simpler, more readable and easier to maintain and, more importantly, easier to prove correct,
- the mapping and traceability between the elements of the user model and the source code resulting from the model transformations would have been easier; in the ASSERT case this factor was paramount, in that the code generation was not integral, it was thus necessary to understand the generated code before adding/integrating additional fragments,
- the modeling space would have been kept intact, thereby simplifying the use of the tool by the users, especially those who did not need to use the additional features.

3 Action Languages and Product Evolution

One of the main advantages in applying the MDA paradigm consists in providing a tool that can also be used by people who do not necessarily have full mastery of the employed technologies: a realistic example of a complete general-purpose system is composed of a back-end application, a graphical user interface and some support for data persistence; the designer should be able to conceive a complete application without necessarily having to be expert with all of the technologies in use in the tool infrastructure and operation.

Implementing a tool for the automatic generation of executable resources is a challenging task: it is often desirable to support a variety of choices from the technological point of view (programming languages, means for data persistence etc.) and, equally often, each generated system requires the combination of a

score of technologies (Web services, databases, resources for the graphical rendering of the applications etc.). In some cases, the set of aspects to be modeled is so wide and/or complex that some of them inevitably remain outside the modeling space and must thus be implemented by hand by domain experts. This situation is in striking contrast with one of the major goals of the model-driven approach, which promulgates that the user should not have anything to do with everything that is automatically generated. What really happens, instead, is that domain experts are frequently consulted for advice on how to customize the system or to inject in it some elements that reflect user experience in the target domain.

A well-known limit of the modeling tools based on graphical editors is represented by the difficulties in modeling the behavioral aspects of the target systems exhaustively, allowing the user to graphically represent all the distinctive features and characteristics of the expected application behavior [SPH⁺01]. The tool that we realized for the ASSERT project, for instance, enables the user to model the structure of the system through a set of diagrams and to express (part of) the desired execution semantics by means of non-functional properties of the model components and state diagrams inspired on UML [OMGb]; the most part of the behavioral characteristics of the user-level entities was however left out of the formal description of the system, thereby excluding an integral generation of source code. This limitation gave rise to at least two undesirable side effects:

- in order to obtain an operating application, the user of the modeling tool must include some code fragments (possibly automatically produced by foreign MDA technology) by hand; this need implies that the user must fully and deeply understand the structure and contents of the generated code and of the mapping between the elements specified in the input model and those produced by the automatic transformations;
- the models employed in the transformations do not cover the application specification exhaustively: the lack of expressive means to model important features like, for example, the communication protocols between threaded components, had a big impact on the usability of the tool. The automatic transformations, thus, are only able to extract partial fragments of information from the models; in ASSERT some of the infrastructural enhancements that we considered were deeply affected by this limitation, which turned out to be a heavy drawback to the possible extensions of the tool.

A possible solution to the problem of *entirely* modeling the execution semantics is to use an action language, that is, a high-level formalism with a sufficient degree of expressiveness to permit full generation of the system starting from the input model; multiple examples can be found in literature in support of this contention, and the Executable UML profile is one of its applications [Mel]. This solution requires a modeling approach that is surely not as intuitive and easy to understand as a purely graphical representation; in some cases, the growth in the number of user-required functionalities can lead to the development of a modeling language with complexity comparable to, and perhaps even greater than, that of a commonly-used programming language, for example, Java.

When a modeling tool reaches a considerable level of maturity and its modeling features become very complex, it can be reasonable and even advisable for the tool designers to ask themselves some questions, just to know whether they eventually held on to their initial goals: Does the tool really make its users' life simpler? Are the advantages brought by all of the supported features really worth the time spent in teaching users how to properly use them? Is this effort compensated by the benefits to be obtained from the model-driven approach?

Arguably, the goal to strive for to obtain a good modeling tool is really to strike the right balance between two important objectives: simplifying the development process of the final product and granting an acceptable level of expressiveness. Most importantly, this balance must be maintained during the evolution of the tool, that is, in the face of growing user expectations.

4 Optimization

Besides having to satisfy increasingly complex requirements for the solution of specific problems, everyone who designs a model-driven development tool must cater for some degree of optimization of the generated code, from both the architectural and the performance standpoints. In fact, even if nobody should ever need or want or bother to look at the source product of the automatic transformations, the users of the generated application expect a degree of execution efficiency (thus discounting elegance of presentation) comparable to the one that they would obtain from a “handmade” solution.

While the result of the transformations is sometimes not satisfying, there is however another side of the coin to consider: an application that was automatically generated is a system that was instantly realized starting from a set of formally-expressed requirements; the designer of a high-integrity real-time system is capable of tweaking configuration parameters as delicate as the execution rate of a task and its execution priority, factors that enormously influence the interaction between the control flows and thus the efficiency of the whole system; in the same way, the designer of an enterprise application can change the database type for data persistence, thus being able to perform tests to determine what technological choice makes her system more efficient. If a modeling tool offers this kind of possibilities, the initial premises are overruled: a model-centric approach can be very useful to optimize the target systems without necessarily involving a development team for the implementation of test prototypes.

As for the ASSERT project, a possible way for optimizing the generated code derives from the model structure imposed by the RCM metamodel. The tool suggests modeling the system on distinct views, seeking to realize virtuous separation of concerns. The two most important modeling views that the ASSERT approach places under user control, are:

- the *Functional view*, in which the behavioral features of the entities of the system are specified;

- the *Interface view*, in which the system functionalities are logically aggregated in application level containers (APLC), which publish them as provided or required interfaces (see [OMGb] for a precise definition of provided and required interfaces); the non-functional attributes, like the access protocols to be applied to the provided services, are defined at this level [Bor06].

A third view is added by the automatic transformation engine: during the model-to-model transformation, new entities, called virtual machine level containers (VMLC) are introduced, which permit to concretely realize in compliance with the RCM the non-functional requirements set forth in the Interface view.

In the RCM methodology, every service invocation is performed on a method interface provided by an APLC; the execution of the method, though, is delegated to the entity that realizes the non-functional properties declared for the APLC at model level, namely a VMLC; the latter, in turn, delegates the implementation of the service to the behavioral specification defined in a functional entity, as Figure 3 shows.

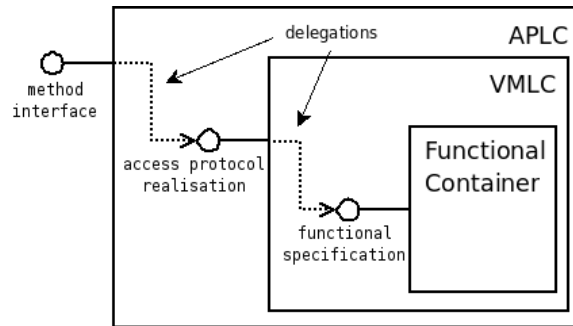


Fig. 3. The delegation structure for a provided interface in the RCM framework.

In the architecture that was designed for the ASSERT tool-set, each container belonging to one of the three levels has its own representation in the generated code (when Ada code is produced, for example, one package is created for each container). This decision implies that the delegation relations associated to provided services are represented in the source code as chain of invocations. The result is a representation that intensively employs the object-oriented paradigm and that also prompts a number of ideas for efficiency-related enhancements. Since the tool is meant to be used mainly for the modeling of embedded, high-integrity systems, the consideration of efficiency aspects assumes considerable importance. Full code-to-model traceability is a bonus that surely helps verification but that pays a toll to efficiency.

A way to mitigate the effects of this problem consists in avoiding the generation of a code representation for the APLC; in this way, the concurrent features of model components would be directly realized by the VMLC (without the APLC

layers around them) whereas the functional entities would continue to be represented by functional containers. Although this solution is simple and effective, it would undermine an important property of the transformation framework: the traceability of mapping between the models (PIM and PSM) and the generated code; in the ASSERT context, in which a sizeable portion of the functional code must be introduced "by hand" by the user after production with ad-hoc foreign tools, this aspect is particularly delicate since the user must thus fully understand the results of the automatic generation.

From this experience, we can thus draw another suggestion concerning the evolution of modeling tools: if some aspects of the generated application must be coded by hand by the user, it is important not to let the optimisation of the generated resources be an obstacle to human intervention, for instance by creating asymmetries between the models and the generated code. It can be useful, therefore, to invest some energy in the task of excluding contacts between the user and the employed technologies as much as possible, so that the optimization issues can take place without causing confusion. Furthermore, as the authors of [MM03] suggest, making the system design explicit also allows one to quickly react to the "hot new technology" effect, which often permits to win efficiency and effectiveness improvements.

5 Conclusions

Is it really preferable to have a modeling tool that adds the complexity of an action language to the graphical editors rather than a totally-graphical one with less expressiveness? From the ASSERT example we learned the importance and the need to explicitly model the execution semantics of the systems. Industrial feedback also taught us that increasing the expressiveness can lead to a tool that is difficult to maintain and use. The control of the tools evolution seems to be a constant issue both in niche domains like the development of high-integrity real-time systems and in the broader context of the applications used for the management of data and for the users interaction. It is thus necessary to define a solid methodology to determine: (1) the indispensable components of any model-driven application; and (2) a set of guidelines to state how the evolution of the tools must be disciplined.

Our experience indicate that the problem does not lie in the presence of an action language but rather in the way the tool evolves in response to the increasing user requirements.

To this end, a possible guideline can be derived from both experiences that inspired this paper: given the need to reconcile the inexhaustible demand for highly-expressive tools with the need to generate efficient, optimized applications, and given the difficulties tied with managing a modeling tool whose complexity is unrelentingly growing, it is important to ponder very carefully on what functionalities must be introduced to yield enhancements to the expressive power of the tool and which instead should be developed as side libraries, by aggregating preexisting features and preserving the simplicity (and the correct-

ness) of the original model transformations. Simplicity does not only mean ease of learning how to use the modeling tool, but also more chances to eventually get to realize a tool that permits to completely exclude direct interaction of the user with the technology. This solution permits to avoid the need to provide native (thus direct) support for functionalities that can otherwise be realized through already-supported features (thus indirect), and it allows to concentrate on the aspects that cannot be presently modeled, so as to avoid their by-hand implementation by technological experts.

Acknowledgments

The experience discussed in this paper was made in the ASSERT project (IST-FP6-2004 004033), partially funded by the European Commission as part of the 6th Framework Programme. The opinions and views which were presented, however, are those of the authors only, and do not necessarily correspond to those of the other members of the ASSERT Consortium.

References

- [BDV03] 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/ftplib/reports/YCS-2003-348.pdf>.
- [Bor06] M. Bordin. Correctness by construction and separation of concerns in the development of high-integrity real-time systems: a metamodel-driven approach. Master's thesis, Dept. of Pure and Applied Mathematics, University of Padua, Italy, 2006.
- [BPV08] M. Bordin, M. Panunzio, and T. Vardanega. Fitting Schedulability Analysis Theory into Model-Driven Engineering. In *ECRTS'08*. IEEE, July 2008. (to appear).
- [Mel] Stephen J. Mellor. Agile MDA. URL: citeseer.ist.psu.edu/652754.html.
- [MM03] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. <http://www.omg.org/docs/omg/03-06-01.pdf>, 2003.
- [OMGa] OMG (Object Management Group). <http://www.omg.org>.
- [OMGb] OMG (Object Management Group). UML 2.0 Superstructure Specification.
- [Pan06] M. Panunzio. Teorie e Strumenti per l'Analisi Temporale di Sistemi Real-Time a Struttura Gerarchica. Master's thesis, Dept. of Pure and Applied Mathematics, University of Padua, Italy, 2006.
- [Pro] Assert project. <http://www.assert-project.net>.
- [SPH⁺01] G. Sunyé, F. Pennaneac'h, W.-M. Ho, A. Le Guennec, and J.-M. Jézéquel. Using uml action semantics for executable modeling and beyond. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, volume 2068 of *LNCS*, pages 433–447. Springer, 2001.