Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

From Complex UML Models to Systematic Performance Simulation

Olivier Constant, Wei Monin, Susanne Graf

Verimag Research Report n^o TR-2007-10

17/09/2007

From Complex UML Models to Systematic Performance Simulation

Olivier Constant, Wei Monin, Susanne Graf

17/09/2007

Abstract

The creation and deployment of a service by a telecommunication operator is a complex activity where functional correctness and performance issues are equally critical. We thus promote a methodology that associates rigorous system definition and performance analysis throughout the entire design process, based on the guaranteed availability of analysis models. In this paper, we report how we applied a Model-Driven Engineering approach to build a tool that supports the methodology. The tool provides automatic and systematic transformations of UML 2 design models of a high degree of complexity to performance models for an industrial simulator. We stress in particular how attempting to build a reliable MDE process required careful design decisions and a thorough preliminary study of syntactic and semantic concerns.

Keywords: Model-Driven Engineering, Model transformation, UML, Semantics, Simulation, Performance analysis

Reviewers: Laurent Mounier

Notes:

How to cite this report:

@techreport { ,
title = { From Complex UML Models to Systematic Performance Simulation},
authors = { Olivier Constant, Wei Monin, Susanne Graf},
institution = { Verimag Research Report },
number = {TR-2007-10},
year = { 2007},
note = { }
}

1. INTRODUCTION

Telecommunication operators have to rapidly deploy new services on existing distributed infrastructures. Those services have to process simultaneous requests from a large number of customers who ask for an increasing degree of sophistication.

To tackle this challenge, development teams usually follow methodologies that are based on semi-formal modeling languages. Formal methods are sometimes applied to provide stronger guarantees on the (functional) design. Nonetheless, the correctness of the software is by no means sufficient to ensure satisfactory quality or cost-effective deployment.

As reported in [8], most information systems are unnecessarily overdimensioned (e.g. useless memory and CPU power) and, worse, the reason for this situation is unknown: possibly unsuitable technical solutions, irrelevant allocation of software units to nodes, etc. For the service provider, this has a very negative impact on the overall cost.

Performance analysis techniques allow studying the behavior of a system in terms of performance, based on a performance model. A performance model is expressed in a specialized language, most of the time an extension of Queueing Networks or Petri nets. Ideally, performance analysis should be carried out from the very beginning of design throughout the entire development process, since the benefits of performance analysis depend on the design stage at which it is performed.

- At the early stages of design, performance analysis provides end-to-end response time and throughput estimations that aim at detecting and preventing problematic design decisions, typically of an architectural nature [19].
- As the design progresses, models get more precise and complex, and so do performance models and analysis results.
- At later stages, when the architecture is fixed and design decisions get precise enough, performance analysis provides indicators, such as resource usage, to estimate what characteristics the resources must have to allow the system to meet its performance requirements. Examples of resource characteristics are the processor power, the memory size, and the network bandwidth. The ultimate goal is to determine an infrastructure that allows the system to meet its performance requirements at an acceptable cost.

Despite these economic advantages, performance analysis is rarely applied in real-life projects. We observed that in the telecommunication world software designers are not familiar with performance analysis techniques: building a good performance model of a large software application is a difficult task that requires skilled experts. In addition, performance experts do not directly participate to the design activity. There is therefore a risk that they have an incomplete understanding of the architecture to be built and elaborate a performance model whose relevance is uncertain.

These observations emphasize the need for a methodology that guarantees a closer coupling between performance analysis and design. A significant number of research proposals, e.g. those surveyed in [3], have investigated approaches where early design models are transformed into performance analysis models. This transformation principle allows reducing the cost of producing models and favors consistency between design and performance models. More generally, a Model-Driven Engineering (MDE) approach is adequate to support a performance/design methodology as shown in more recent publications [12, 21, 18, 2, 9].

Applying such an approach in an industrial context, however, faces additional constraints: engineers should keep using the commercial tools they are productive with, the transformation process should be reliable and scale to complex, realistic models, etc. Obviously, these considerations have an impact on the application of MDE techniques and principles.

This paper is structured as follows. The methodology we promote is sketched in Section 2. Section 3 then describes how we derive a certain number of requirements from the methodology and how they drive the design

of a dedicated MDE process. Section 4 provides an example that illustrates the features we support and the MDE process. Finally, Section 5 provides an overview of the related work.

2. PROMOTED METHODOLOGY

The methodology we promote has been introduced in [4] and has been developed in the Persiform project [17]. Its main objective is to enforce an effective coupling and to favor consistency between software design and performance engineering *throughout the entire design process*. Nonetheless, it also aims at being realistic and pragmatic based on our experience in the telecommunication industry.

The methodology is based on the use of UML diagrams for architectural design: Use Case diagrams for identifying the main functions, Activity or Sequence diagrams for the description of the use case behaviors, Deployment diagrams for the specification of deployment and infrastructure resources. Resources are shared elements such as CPUs, memories, thread pools, network links. Other diagrams can be added as well. The resulting functional model is classically produced in the design flow independently of performance analysis.

It is also a starting point for deriving a model for performance analysis. Nevertheless, the functional model must be manually *adapted* and *enriched* first (see Figure 1). We observed that manual adaptation was often necessary because the rationale that drives functional design is not the same as the rationale for performance modeling. The former focuses on application logic and the latter on resource consumptions.

The adaptation consists in:

1. Ensuring that the set of use cases being modeled is both consistent and relevant for performance. Relevance means that every use case considered has an impact on performance that is worth studying. Consistency means that if a resource is used significantly for the realization of a use case, then all use cases that also use the resource significantly must be modeled.
2. Ensuring that the use case behaviors are described in a way that is suitable for resource consumption specifications. This may imply to group or decompose the actions so that 1) every action that is resource-consuming can be characterized by an actual resource consumption (e.g. unitary processing time) by metrology for pre-existing software elements or else by estimation, and 2) resource consumptions are of homogeneous granularity.

Then the enrichment of the resulting model follows. It consists in actually adding resource consumptions and other performance-related information. This information includes resource-independent durations and timing, resource capacity, and expectations about the environment of the system. This kind of enrichment is classical and promoted by the authors of the aforementioned works and by the existing UML profiles for performance [14, 13, 7].

Obviously, the construction of this “performance-oriented” design model requires the designer to have at least some knowledge of performance concepts. Should this not be the case, the designer may be assisted by a performance expert.

At this point, the performance-oriented design model is *automatically transformed* into a performance model expressed in the specialized formalism of some performance analysis tool. This is where a performance expert takes over. He carries out a performance study based on this performance model. This usually involves experimenting with different parameters, configurations and environments.

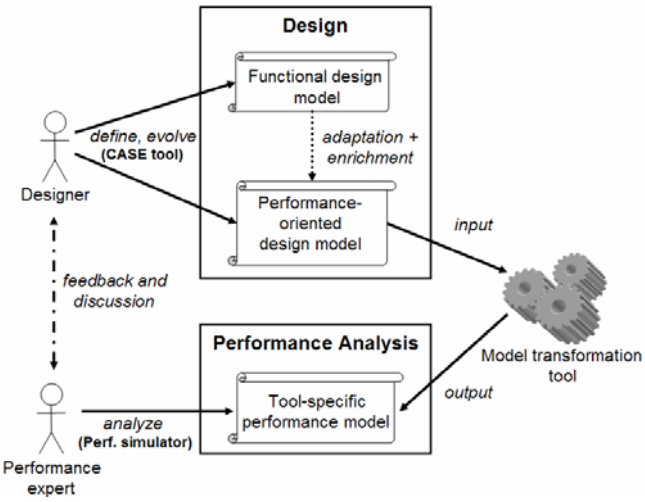

Figure 1. Principles of the Persiform methodology

Feedback is eventually provided by the performance expert to the designer (note that those are roles that can be played by the same person, only skills are concerned). The impact of design decisions is criticized based on performance criteria, which may motivate the designer to reconsider some decisions and modify the functional design model.

Whenever this model is modified for performance reasons or refined in the normal course of design, a new iteration of the process starts. The "performance-oriented" design model typically evolves in parallel to the functional model.

Note that this methodology does not suggest trying to automatically construct a performance-oriented design model, or enforce permanent synchronization of design and analysis models. Neither does it intend to bypass performance experts by automating analysis and feedback. It focuses on continuously taking performance feedback into account during the entire design process, even when models become complex.

Still, we are convinced that the consistent, automatic generation of a full performance model as a starting point for performance analysis is a significant improvement over ad-hoc performance models created from scratch and with a limited understanding of the system under development. It is an improvement in terms of confidence and productivity, but also in terms of the effectiveness of the collaboration between the two actors, designers and performance experts.

Both actors are enabled to argue about models that consistently share a common structure and terminology, which is the main achievement. Nonetheless, each actor keeps using his own favorite, specialized industrial tool. We believe this point to be crucial for cultural reasons and productivity. In particular, there exist few performance analysis tools that are robust and mature.

The choice of the performance analysis technique is clearly not neutral since its scalability is vital to the applicability of the methodology. This pleads for discrete-event performance simulation. This technique is relevant also because it is supported by well-known and user-friendly commercial tools such as OPNET [16] or HyPerformix Workbench [10] that are used in various companies.

Alternatively, analytical methods are more efficient and it would be beneficial to apply them whenever possible. Nevertheless, they can only be applied to models satisfying strong hypotheses so they become less suitable at advanced stages of design.

3. MODEL TRANSFORMATION PROCESS

Model transformation is at the core of the methodology, and therefore the definition of an adequate tool support is essential.

3.1 Preliminary Requirements

The methodology is not likely to be accepted if the actors cannot trust the design-to-performance model transformation. Whenever a design model is created, it must be possible to check that the model satisfies a set of constraints imposed by the methodology. If so, it must be guaranteed that the model will *actually* be automatically transformed into a *faithful* analysis model; otherwise the tool must provide a diagnostic identifying the origin of the problem.

Let us translate these requirements into the MDE world. In a MDE approach, a model is a piece of structured data that conforms to an abstract syntax specified by another model called metamodel. In addition, a model has to comply with static semantics rules that are associated to its metamodel. A metamodel is usually defined in MOF or UML while static semantics rules are specified in OCL.

In this setting, a model transformation *Tr* is a total function $S_3 \rightarrow T_0$ (see Figure 2), where $S_3 \subseteq S_0$ and S_0 (resp. T_0) denotes the set of models that conform to the source metamodel *S* (resp. target metamodel *T*). Further, we have $S_3 = S_0[R]$ where *R* is a set of static semantics rules and $S_0[R]$ is the set of all models in S_0 that comply with all rules in *R*. *R* actually represents the preconditions of the transformation.

According to our requirements, *R* has to be explicit in order to allow an OCL tool to check the applicability of *Tr* on a given model. Further, we want our transformation to be *systematically applicable*, that is *Tr* must be applicable on every syntactically correct model (w.r.t. abstract syntax and static semantics). In other words, we want $S_1 \subseteq S_3$ where S_1 is the set of source models that comply with the static semantics of metamodel *S*.

Besides being applicable, the transformation has to be correct. This implies $T_3 \subseteq T_1$ for the *syntactic correctness* of the models generated. In addition, it is fundamental that the transformation preserves the semantics of models in order to obtain meaningful analysis results We mean here the *dynamic* semantics, which in our case is operational for simulation purposes.

If models are allowed to have an undefined or ambiguous semantics, then their analysis is hazardous because the semantics assumed by the analysis tool may be different from the intended one. This is even more critical when several transformations to different analysis tools are used. Every model in our approach should thus have a non-ambiguous, well-defined semantics.

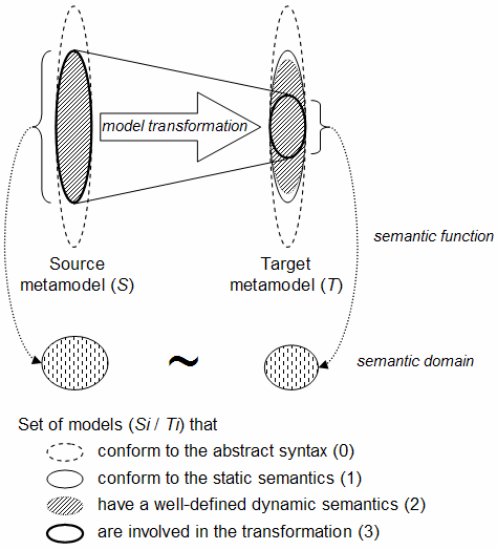

Figure 2. Systematically applicable, semantics-preserving model transformation

Ideally, "semantic" metamodels and associated static semantics rules would be designed such that every conformant model has a well-defined dynamic semantics. In practice, however, defining an exhaustive set of static semantics rules may not be trivial. Hence, to be *semantically meaningful* a model transformation must guarantee that $S_3 \subseteq S_2$ and $T_3 \subseteq T_2$, where S_2 and T_2 are the subsets of S_0 and T_0 collecting the models which have a well-defined dynamic semantics.

Then, such a model transformation must be *semantically correct*, which in our case means that it is semantics-preserving. In other words, given that D_S and D_T are the semantic domains of S and T, and we have the semantic functions $I_S : S_2 \rightarrow D_S$ and $I_T : T_2 \rightarrow D_T$, then we must have $\forall m \in S_3$, $I_S(m) \sim I_T(Tr(m))$, where $\sim$ denotes an equivalence relation. For example, from an operational perspective a source model and the corresponding target model must yield equivalent sets of execution traces.

Figure 2 represents a model transformation that is systematically applicable and semantically meaningful on the source side ($S_3 = S_1 \subseteq S_2$), and syntactically correct and semantically meaningful on the target side ($T_3 \subseteq T_1 \cap T_2$). Semantic preservation is illustrated too. Hence, Figure 2 can be seen as a representation of the overall transformation process we need for our methodology, where S is the metamodel of design models and T the metamodel of performance models.

Although rather straightforward, these considerations are essential guiding principles for the actual design of our transformation process.

3.2 Design: Conceptual View

The principles above are related to a coarse-grained, ideal view of the overall design-to-performance transformation process. When shifting to an actual design, additional considerations and constraints arise.

3.2.1 Target side (WB metamodel)

On the target (performance) side, the analysis language usually has a precise dynamic semantics and supports static semantic checks. The main risk is lack of expressiveness: is the language flexible enough to accurately and conveniently represent all the concepts from the source side?

According to the principles of our methodology, we have chosen a performance simulator that is used by the industrial partners of the project, HyPerformix Workbench. Its language includes high-level customizable constructs based on Queueing Networks, but also lower-level constructs close to the C language which allow representing sophisticated user-defined concepts.

We have created a metamodel *WB* that models a subset of the Workbench language. Since Workbench models can be checked, compiled and executed, we consider this metamodel to be well-defined in terms of static and dynamic semantics (see Figure 3) even though the latter is not specified formally.

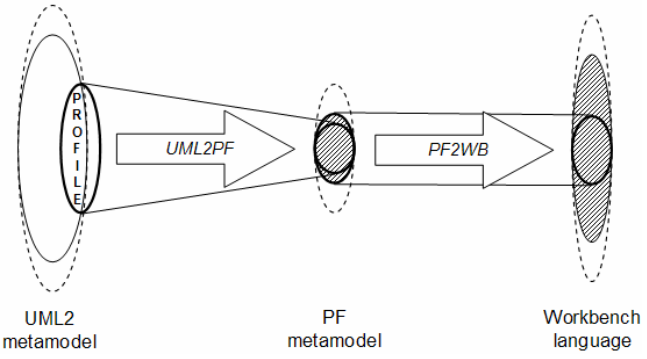

Figure 3. Conceptual view of the transformation process

3.2.2 *Source side (UML profile)*

The situation is less simple on the source (design) side because UML 2 must be specialized for our methodology. This is allowed by the profile mechanism which supports the extension of predefined concepts via stereotypes and the addition of restrictions specified as OCL rules of static semantics. The inevitable counterpart of this flexibility is that UML cannot have a complete precise dynamic semantics, but it is up to methodologies and associated profiles to provide one.

We have therefore defined a dedicated profile *PF/UML* (PersiForm/UML) that extends UML2 for specifying time, environments, and the consumption, definition and deployment of resources such as processors, memories, networks. This part does not raise any particular issue since similar concerns are already covered by OMG profiles. Most of the difficulty is actually on determining what extensions and constraints are needed for models to be transformable and have an unambiguous semantics.

For this purpose, the definition of an intermediate metamodel has several benefits, see e.g. [5, 21]. Besides making the transformation process extensible to additional design and analysis languages, it helps make it reliable. For this we need an intermediate metamodel *PF* (PersiForm) that is such that:

- It has a clear operational semantics.
- It allows defining a transformation *UML2PF* from PF/UML. This transformation actually defines the semantics of PF/UML.
- It provides high-level concepts that allow generating well-structured, readable analysis models. This is essential to the relevance of analysis results.
- It remains simple and focused on the concepts of the methodology.

Hence, the overall transformation process has to be refined as the composition of two model transformations with different roles. The first one, systematically applicable on PF/UML, defines a semantics while the second, systematically applicable on PF, is semantic-preserving. Note that, as represented in Figure 3, the composition of these two transformations fulfills the requirements stated in the preceding subsection and illustrated in Figure 2.

3.2.3 *Intermediate metamodel (Petri nets)*

In order to have an intuitive operational semantics, the PF metamodel is built upon concepts as they exist in two well-known frameworks: Petri nets and Queueing Networks. Petri nets are convenient for expressing concurrency and synchronization, while QNs allow representing resource consumptions in a high-level manner. We need both frameworks in order to be at the same time expressive and high-level.

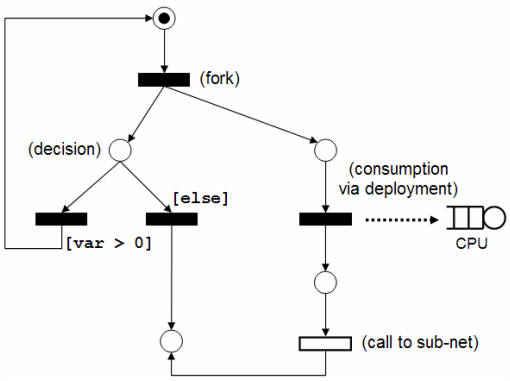

Figure 4. Informal representation of a PF model

The behavioral part of PF models is a set of hierarchical (“procedural”) unsafe Petri nets, where resource consumptions are special transitions that refer to resources specified by queueing stations (see Figure 4). PF Petri nets may have datatyped parameters and variables. Transitions may take time, manipulate variables or call sub-

Petri nets. Behaviors can involve deterministic as well as probabilistic decisions thanks to classical probability distribution functions.

This metamodel is quite rich, but this is necessary to cover UML Activity diagrams with data and resource consumptions. Static semantics rules restrict PF by forbidding a few unwanted Petri net patterns, such as transitions mixing time, logical constraints and synchronization.

3.2.4 Issues met

Unsurprisingly, such a rich set of concepts raises semantic concerns. For the transformation process to meet its dependability requirements, these concerns need be systematically addressed.

In particular, we met issues regarding the management of concurrency. UML2PF semantically translates a UML2 activity to a PF Petri net (PFPN). UML activities and PFPNs are "procedural": they may have several concurrent executions due to concurrent calls. In PF, executions correspond to disjoint sets of tokens. We assume the absence of cycles in the call graph, which forbids recursive calls. UML Activity diagrams do not impose particular restrictions on patterns such as those in Figure 5. Unless we have a good reason to forbid them, such patterns must then be accepted and have an adequate semantics.

Nonetheless, in performance analysis the notion of causality is fundamental, e.g. for determining end-to-end response time. Threads of causality naturally correspond to tokens. Hence, we want only tokens belonging to a same execution of an activity/PFPN to be able to synchronize in a Join node, otherwise causality threads may get lost. A naïve mapping from Activity diagrams to Petri nets would not be acceptable because in Petri nets tokens are not distinguishable.

Further, let us consider the third pattern in Figure 5: each iteration of the loop creates a new token that executes a Fork/Join. Since the models are stochastic, it is possible that tokens from different iterations of the loop arrive in the Join node at the same time. Should they join or not?

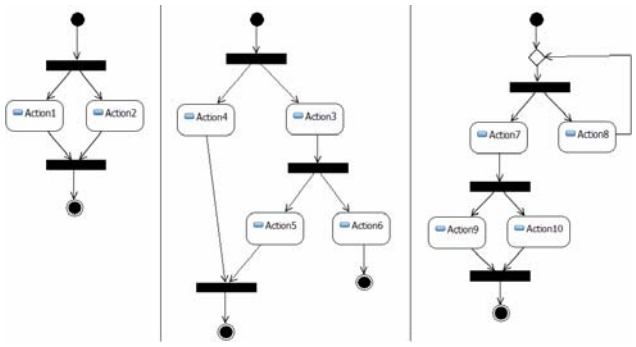

Figure 5. Patterns and causality in Activity Diagrams

In order to be able to forbid or allow such tokens to join, PFPNs are actually defined as colored Petri nets that make a very limited usage of colors. We introduced some simple syntactic extensions in Activity diagrams which allow users to disambiguate the models when exceptionally needed. For example, users have the possibility to specify that a Join node should ignore the color (that is the origin) of the tokens that come through a given incoming arc.

Our semantics also covers special final nodes introduced in [4] that bring more flexibility to the specification of sub-activity termination. For that purpose colors of tokens are defined as stacks of identifiers, as can be found in deliverables of the Persiform project [17].

Other issues come from the mix of synchronization and data. A UML activity is a namespace in which variables (attributes) may be defined. Do all tokens "see" the same values for these variables? This would be restrictive because we need variables that are local to tokens to specify e.g. looping conditions. Consequently, we distinguish (using stereotypes) between variables of kind *shared* and *flow*. Each token has its own copy of flow variables, while shared variables are common to all.

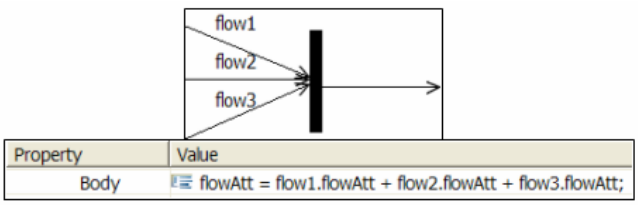

Figure 6. Specifying the merge of local data in a Join node

Nevertheless, this, in turn, raises an issue in Join nodes. Since flow variables are private to tokens, how to define the values of the flow variables in a resulting, merged token? We decided to let users define values in Join nodes explicitly by referring to incoming arcs, which in turn implicitly refer to incoming tokens. Figure 6 provides an illustration where the flow variable *flowAtt* of the resulting token is defined to be the sum of the values of *flowAtt* from the tokens being merged.

3.3 Design: Technical View

So far we have essentially dealt with the languages (metamodels) and modeling concepts at stake. We have considered two different concerns: for a given model specified in a user language,

1. give it a clear dynamic semantics;
2. translate it correctly to an analysis language.

We call the transformations that treat these concerns, UML2PF and PF2WB, “semantic-driven”.

The transformation process needs to be further refined in order to deal with issues that are exclusively technical, i.e. unrelated to modeling concepts. The main issue is to interface with the analysis tool. Hence the following additional concerns:

3. organize the content of the model to match the input format of the analysis tool, if applicable;
4. turn the model into an actual document.

Point 4 simply consists in extracting text from a model and saving it into a file. Point 3 is only relevant if the input format of the analysis tool is based on a meta-syntax, typically XML. Instead of directly outputting an XML document complying with the Schema or DTD of the analysis tool from a WB model, it is possible to proceed in two separate steps.

A first model transformation *WB2XML* translates the high-level WB model into a model conforming to a metamodel of XML (Point 3), then a second, trivial transformation *XML2Text* outputs an XML document from the XML model (Point 4). These transformations are represented in Figure 7. They are called “technical” by contrast with the semantic-driven ones. Note that XML2Text could be replaced by a predefined XML “extractor” such as the one provided in [1].

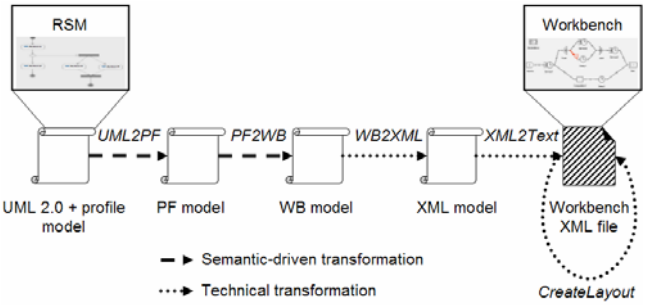

Figure 7. Technical view of the transformation process

At this stage, our transformations allow UML2 models to be transformed into XML files that can be opened and compiled with Workbench. This is however not quite satisfying because the generated files cannot be effectively used by performance experts. Since Workbench models are graphically represented in the GUI as graphs, layout information needs be added to model elements. Hence the last concern:

5. generate a layout for the graphical display of the model in the analysis tool, if applicable.

This corresponds to a last transformation *CreateLayout*. This transformation is represented in Figure 7 as defined on XML files: this is because we implemented it so for pragmatic reasons, but it could be a classical model transformation.

3.4 Implementation

In the previous subsections, the overall transformation process has been defined as a composition of smaller transformations according to a separation of concerns, involving intermediate metamodels. Unsurprisingly, this modular design helps develop an implementation that is simple and readable to a certain extent.

Since model transformations are functions by essence, we wanted to specify them at the highest possible level of abstraction, in a way that would ease carrying out informal proofs. They have been finally implemented in purely declarative ATL.

ATL (Atlas Transformation Language [11]) is a hybrid model transformation language that, for its declarative part, is close to OCL. An ATL transformation is actually a set of functions called *matched rules* that associate a set of elements conforming to the target metamodel to a certain type of elements of the source metamodel. That type is specified as a metaclass and a constraint. When the transformation is executed, new elements of the target metamodel are created for every element conforming to the type.

For example, Figure 8 shows a matched rule from UML2PF which specifies that for every edge in Activity Diagrams such that constraint `mustProducePlace` holds, a PFPN place is created. `mustProducePlace` is further defined in Figure 9 as a boolean query on activity edges.

Predictably, the biggest transformation turned out to be PF2WB: it is the semantic-preserving translation which is the very core of the transformation process. With PF made of 36 concrete metaclasses and 31 rules of static semantics and WB made of 39 concrete metaclasses, PF2WB contains 1700 lines of code. This includes about 500 lines of Workbench C code used to translate complex concepts.

Overall, the maintainability and extensibility of the implementation have been successfully tested during the Persiform project. We believe that structuring the implementation as a set of declarative ATL transformations was actually beneficial. The transformations will be available on the Persiform web site.

```
rule Edge2SimplePlace {
    from e : UML!ActivityEdge
        (e.mustProducePlace())
    to
        p : PF!SimplePlace (
            name <- e.name.newPlaceName(),
            net <- e.getPNProducer(),
            isColoring <-
                if (e.hasStereotype('PFArc')) then
                    e.stereotypeValue('PFArc', 'isColoring')
                else
                    false
                endif
        )
}
```

Figure 8. An ATL matched rule in UML2PF

```
helper context UML!ActivityEdge def:
mustProducePlace() : Boolean =
    not self.producesTransition()
        and
    self.source.transformsToTransition()
        and
    self.target.transformsToTransition();
```

Figure 9. An ATL helper in UML2PF

Transformations apart, the tool we have developed has a few limitations regarding our requirements. We did not achieve the definition of an exhaustive set of static semantics rules for the PF/UML profile. A number of such

constraints have been defined but the size and complexity of UML2 make this task difficult. In addition, we do not parse or check the textual expressions that designers are allowed to write in design models (e.g. guards on conditionals). Those expressions are kept unchanged and passed as strings throughout the transformation process and finally compiled with Workbench. As a result, errors in expressions are detected late.

To validate our transformation tool, we have compared simulation outputs of a legacy performance model built "from scratch" with outputs resulting from the application of the entire transformation process on a UML model of the same system. Both outputs turned out to be identical. Of course, more similar experiments would provide more trust.

4. EXAMPLE

This section illustrates how systems can be modeled and transformed in our approach. The original case study that drove the Persiform project included 17 parameterized activity diagrams. For the sake of clarity, a simpler example is shown here inspired by a real system at France Télécom. It has been modeled with the UML CASE tool Rational Software Modeler (RSM).

The system, in this example, provides an audio conference service. It allows customers that are identified in a database to organize and participate to audio conferences. An organizer registers an audio conference at a given date and time for some duration, with certain participants and a leader. An audio conference starts when the leader has joined. The realization of an international conference requires a larger amount of resources than a national one.

The Use Case Diagram on Figure 10 identifies 2 actors, *Organizer* and *Participant*, and 2 corresponding use cases. Use cases can be enriched with information about system usage thanks to the PF/UML profile. When using RSM, clicking the actor *Participant* pops-up the box on the right side of Figure 5. Here, the designer specified that one request from a participant is uniformly received by the system every 0 to 5 seconds.

As every use case is associated with a behavior that may have formal parameters, actual parameters are passed via the *passedValue* property of the actors. In the example, the behavior of *ParticipateToConference* has a single boolean parameter named *conferenceIsInternational*. The *passedValue* property of actor *Participant* contains one entry: the boolean expression `Bernoulli(0.10)` which states that 1 conference out of 10 is international.

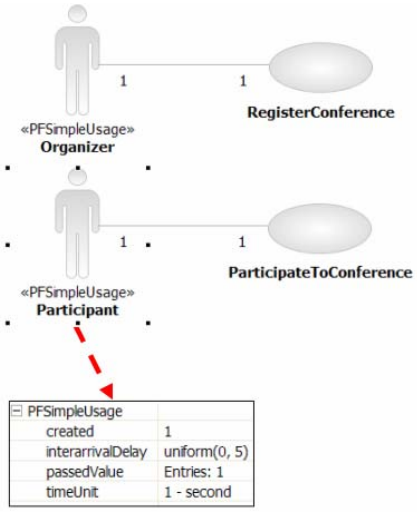

Figure 10. Use case diagram: characteristics of the environment

A Deployment Diagram represents the infrastructure and the resources of the system (see Figure 11). The system is deployed on 2 nodes (machines) named *MainHost* and *Database*. The first one is specified to have 2 processors and to manage its tasks according to a PS (Processor Sharing) discipline without priorities or preemption

In terms of deployment units, the system is structured into 3 artifacts: one dedicated to the business logic of the *Service*, one managing a *VocalPlatform*, and a last one that contains the *DatabaseManager*. The latter is deployed on the *Database* while the others are deployed on the *MainHost*. In addition to processors, the PF/UML profile allows specifying memories, passive resources such as threads, and network links characterized by a latency and bandwidth.

The behavior of the system is defined by Activity Diagrams. The PF/UML profile restricts UML2 activities to control flows by guarded alternatives, arbitrary synchronization, synchronous calls to sub-activities and to operations on artifacts. We aimed a subset that is consistent albeit expressive. Activities may have datatyped parameters and attributes. Delays, resource consumptions and data manipulations are specified in actions. Data expressions and assignments are specified as strings complying with the usual C/Java syntax for booleans and integers.

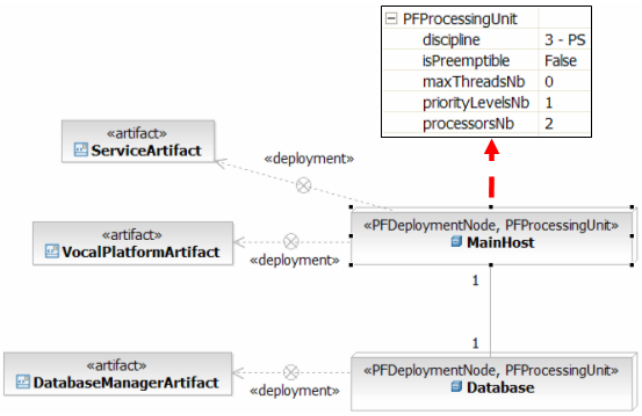

Figure 11. Deployment diagram: characteristics of the resources

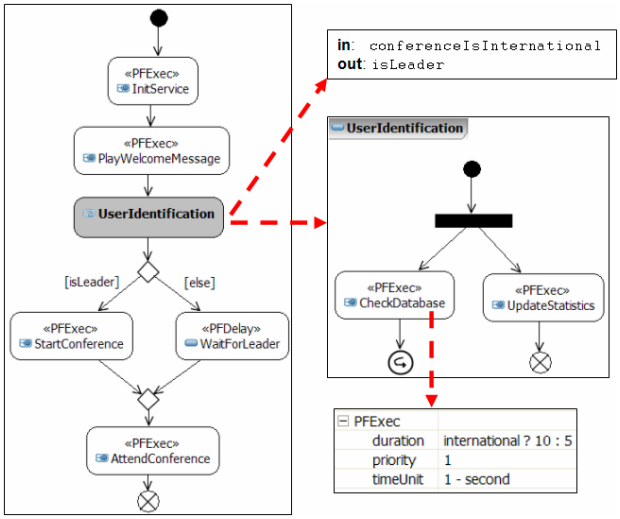

Figure 12. Activity diagrams: behavior decomposition and resource consumptions

Figure 12 shows the behavior associated to the *ParticipateToConference* use case: it includes several actions and a deterministic alternative based on the attribute *isLeader*. This attribute is assigned the 'out' value of the call to sub-activity *UserIdentification*.

The sub-activity *UserIdentification* has an 'in' parameter *international* and an 'out' parameter *leadership*. When the sub-activity is called, it executes the *CheckDatabase* action which consumes 10 or 5 seconds of processing

time depending on the value of *international*. The consumption implicitly applies to machine *Database* because the action is associated to *DatabaseArtifact*.

In parallel, an action *UpdateStatistics* is executed. The activity returns when *CheckDatabase* is completed. This is done independently of the other action but without interrupting it. Since such a semantics does not exist in UML2, we have added a special *ReturnNode* symbol in the PF/UML profile as shown in Figure 7.

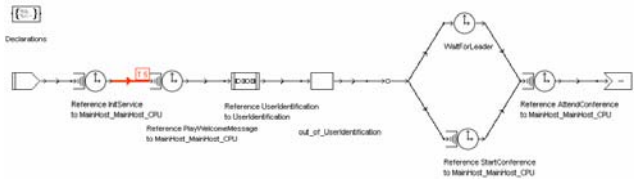

Figure 13. The resulting performance model being executed with Workbench

The behavior associated to the other use case is specified similarly. When the designer is done, he can check the OCL constraints of the PF/UML profile within the UML CASE tool. The model presented here passes the test, so it can be transformed automatically. Figure 13 shows a partial view of the resulting Workbench model at execution time. Except for overlapping text boxes, the graphical appearance is the one that is actually obtained. The model can be compiled, executed and studied with Workbench.

5. RELATED WORK

A number of publications are dedicated to performance analysis by transformation of design models. Some of them are surveyed in [3], others appeared since then [12, 21, 18, 2, 9]. These approaches generally focus on the early stages of design and target analytical methods, so the behavioral aspects of models are more restricted than in our case. To our knowledge our work is the only one that aims at exploiting the flexibility and expressive power provided by a commercial simulator.

Also, few of these approaches mention the issue of the systematic treatment of models and subsequent semantic difficulties. Formal semantics is rarely invoked, with the notable exception of [12] where Activity Diagrams are translated to stochastic Petri nets. In a more general setting, the importance of rigorous static semantics and OCL constraints in MDE is stressed e.g. in [20].

Some of the aforementioned work is based on OMG profiles, in particular SPT. Our profile is different from OMG profiles like SPT or MARTE because its rationale is not the same. OMG profiles are designed to provide a common notation based on a unified domain model. Providing a formal semantics or defining precise rules of static semantics is left to more specific approaches and methodologies like ours.

Note that our models do not entirely fall under the domain model of MARTE since we allow deterministic (non-probabilistic) choices. Nevertheless, our profile could theoretically reuse a subset of MARTE, but it does not because MARTE was not available early enough for Persiform.

6. CONCLUSIONS

We have presented a methodology and a dedicated model transformation process for performance analysis of complex UML2 models based on commercial tools. The methodology allowed deriving requirements on the transformation process. We have justified and illustrated how making the process reliable required the rigorous study of semantic issues and the provision of a semantically rich intermediate metamodel.

Using current MDE technologies, we implemented our transformation process as a composition of simpler, fully declarative transformations. We believe that it allowed us to make our tool maintainable and reasonably extensible. The tool has been tested on large examples as well as borderline cases. It will soon be available on the Persiform web site. A more significant validation phase at France Télécom is planned.

The main difficulty we have met is that we can currently not make sure that a model transformation to a target metamodel *T* actually generates models conforming to *T* if static semantics is concerned. We chose a declarative transformation language in order to give us some confidence and at least ease the specification of informal proofs. Nevertheless, the use of a dedicated prover might bring high benefits. Current research on formal proof with OCL [6] may be a first step towards such an achievement.

7. ACKNOWLEDGMENTS

The authors would like to thank Marius Bozga for countless helpful discussions.

8. REFERENCES

[1] AM3 Home Page (Atlas MegaModel Management). http://www.eclipse.org/gmt/am3/

[2] D'Ambroglio, A. 2005 A Model Transformation Framework for the Automated Building of Performance Models from UML Models. Proc. 5th ACM Workshop on Software and Performance (Palma, Illes Balears, Spain, July 12 - 14, 2005).

[3] Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M. 2004 Model-based performance prediction in software development: a survey. IEEE Trans. on Software Eng., vol.30 (5), May 2004, pp. 295-310.

[4] Bozga, M., Combes, P., Graf, S., Monin, W., Moteau, N. 2006 Qualification d'architectures fonctionnelles. Proc. NOTERE Conf. (Toulouse, France, June 2006).

[5] Bozga, M., Graf, S., Mounier, L. 2002 IF-2.0: A Validation Environment for Component-Based Real-Time Systems. Proc. Intl. Conf. on Computer Aided Verification (CAV 02), LNCS.

[6] Brucker, A.D., Wolff, B. 2002 HOL-OCL: Experiences, Consequences and Design Choices. UML 2002: Model Engineering, Concepts and Tools. LNCS 2460, Springer-Verlag.

[7] Espinoza, H., Medina, J., Dubois, H., Gerard, S., Terrier, F. 2006 Towards a UML-based Modeling Standard for Schedulability Analysis of Real-time systems. Proc. MARTES Workshop (Genova, Italy, October 2, 2006).

[8] Gartner Group, Meta Group and Giga Information Group.

[9] Grassi, V., Mirandola, R., Sabetta, A. 2007 A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. Proc. 6th ACM Workshop on Software and Performance (Buenos Aires, Argentina, February 5-8, 2007).

[10] HyPerformix web page. http://www.hyperformix.com

[11] Jouault, F., Kurtev, I. 2005 Transforming Models with ATL. Proc. Model Transformations in Practice Workshop at MoDELS (Montego Bay, Jamaica, 2005).

[12] Lopez-Grao, J.P., Merseguer, J., Campos, J. 2004 From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering. Proc. 4th ACM Workshop on Software and Performance.

[13] Object Management Group. 2007 A UML Profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems), Beta 1. Document ptc/07-08-04.

[14] Object Management Group. 2005 UML Profile for Schedulability, Performance, and Time Specification v.1.1. Document formal/05-01-02.

[15] Object Management Group. 2007 Unified Modeling Language: Superstructure version 2.1.1. Document formal/07-02-05.

[16] OPNET web page. http://www.opnet.com

[17] Persiform: French RNRT national project. http://www-persiform.imag.fr/

[18] Petriu, D.C., Woodside, C.M., Petriu, D.B., Xu, J., Israr, T., Georg, G., France, R., Bieman, J., Houmb, S.H., Jürjens, J. 2007 Performance Analysis of Security Aspects in UML Models. Proc. 6th ACM Workshop on Software and Performance (Buenos Aires, Argentina, February 5-8, 2007).

[19] Smith, C.U., Williams, L. 2001 Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley.

[20] Wahler, M., Koehler, J., Brucker, A.D. 2006 Model-Driven Constraint Engineering. Electronic Communications of the EASST.

[21] Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J. 2005 Performance by unified model analysis (PUMA). Proc. 5th ACM Workshop on Software and Performance (Palma, Illes Balears, Spain, July 12-14, 2005).