Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 207 (2008) 171–186
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Barbed Model–Driven Software Development: A Case Study

Carlo Montangero [1,2] Laura Semini[2]

Department of Informatics
University of Pisa
Pisa, Italy

Abstract

When thinking of MDE, the immediate understanding is that models drive software development, in the sense that the software is constructed by transforming models from higher levels of abstraction to the point where we reach a model which is executable with the desired degree of quality characteristics. What tends to be less evident, is that, precisely in order to reach the desired quality, many other models are used in the verification and assessment of the solutions under consideration at the various stages of development. That is, looking at the development process, besides a spine of model transformations moving from highly abstract, domain related models down to concrete platform related models (programs), we can see a number of barbs, relating models in the spine to specialized models that permit specific analysis of parts of the software.
In this paper we report on some preliminary work on understanding Barbed Model–Driven Software Development. We are taking an experimental attitude, designing and implementing a barb, using specific technologies and verification tools. The goal is twofold: to get acquainted with the technologies, and to provide a first assessment of their suitability for subsequent explorations. In the experiment experiment the barb deals with the verification of properties of a SOA system modelled in UML.

Keywords: SOA, MDE, UML, VIATRA.

1 Introduction

Models have emerged as primary artifacts of the software development, and model–driven development refers to those software development processes that are based on the use of models and model transformations.

Introducing a recent special issue devoted to Model–Driven Engineering (MDE [3]) [26], D. Schmidt suggests that Model-driven engineering technologies of-

[1] This work is partially supported by the EU project SENSORIA IST-2005-16004. The first author was also partially supported by the Joint PhD program: Department of Informatics, University of Pisa and International Institute for Software Technology, United Nations University, Macao. Thanks to J. Vital, T. Baptista, G. Camacho and M. Jùlio for implementing the system, and to the anonymous reviewers for valuable comments on the first version of the paper.

[2] Email: {cmontangero,semini}@di.unipi.it

[3] MDE is best used to refer to general model engineering ideas, without committing to strict OMG standards: the Object Management Group holds trademarks on MDA (Model Driven Architecture), as well as

doi:10.1016/j.entcs.2008.03.092

fer a promising approach to addressing the complexity of platforms and express domain concepts effectively. The purpose of this note is to highlight an aspect of MDE that may deserve more attention than it has received till now. The benefits we expect from focusing on this facet are in a smoother integration of the analysis tools, based on model-checkers, theorem provers, etc., which are being developed for early static verification. The point is that usually, besides a spine of model transformations moving from abstract models down to programs, there are usually a number of *barbs* that introduce specialized models. These models permit specific, often very sophisticated, analysis of parts of the software under development, usually in the early stages. We can call this approach Barbed Model–Driven Software Development (BMDSD).

In this paper we report on some preliminary work on understanding how much, and how, the barbs should influence the structure of the spine, i.e. how much the models introduced for analysis should influence the structure of the overall development. We are taking an experimental attitude, designing and implementing a specific barb, using specific technologies and verification tools. The goal is twofold: to get acquainted with the technologies, and to provide a first assessment of their suitability for subsequent explorations.

The structure of the paper is the following. The next section presents the issues we are interested in, with respect to BMDSD. Section 3 reviews the technologies we are using in our experiments, namely the profiling capabilities of UML2 [24] as supported by Rational Software Architect [18], the transformation framework VIATRA [6], and the SENSORIA CaseTool [11]. This section also provides motivations for the appeal of these technologies, besides the fact that they are supported by the project we are working in, namely SENSORIA [4]. Section 4 presents the case study, which is centered around Politically Correct [19], a tool that checks statically some security properties of distributed services, exploiting the verification techniques of $\lambda^{\mathtt{req}}$ [9]. Section 5 illustrates a concrete example. After a discussion of related work in section 6, we wrap up with some final considerations.

2 Barbed Model–Driven Software Development

When thinking of MDE, the immediate understanding is that models *drive* software development, in the sense that software is constructed by transforming models from higher levels of abstraction to the point where we reach a model which is executable with the desired degree of quality characteristics. What tends to be less evident, is that, precisely in order to reach the desired quality (both functional, e.g. correctness, and non–functional. e.g. performance, security, usability, etc.), many other models are used in the verification and assessment of the solutions under consideration at the various stages of development. That is, looking at the development process, besides a spine of model transformations moving from highly abstract, domain related models down to concrete platform related models (programs), we can

several similar terms including Model Driven Development (MDD), Model Driven Application Development, Model based Application Development, Model Based Programming, and others.

see a number of barbs, relating models in the spine to specialized models that permit specific, sometimes very sophisticated, analysis of parts of the software. What we have in reality is a *barbed* development process.

Without explicit recognition, the BMDSD idea has been around for some time, and pursued in major projects, like AGILE [1], DEGAS [2], and SENSORIA. In the latter, several barbs are being developed to permit early analysis to guarantee a better quality of the products, in this case with a Service Oriented Architecture.

There are interesting issues with respect to BMDSD in general, related to understanding how much, and how, the barbs should influence the structure of the spine, i.e. how much the models introduced for the analysis should influence the structure of the overall development.

The first issue is related to the technologies to be used in building the development environment, the others are more conceptual in nature, since they are related to the number and kind of models to be introduced in the development process.

With respect to the technologies, the issue is that they should be such as to ease both the vertical development (along the spine) and the integrations of the tools in the barbs. In this paper we experiment with UML2 as the modelling notation and VIATRA [6] as transformation engine to develop a barb for the static verification that a service oriented application satisfies given policies. We think that UML is a good candidate for BMDSD, thanks to the wealth of modelling approaches it supports, most of which are based on well established and popular concepts, and can be fruitfully exploited in the main development. The extension capabilities of UML have already proved able to support the integration of barbs [14].

On its side, VIATRA has built-in facilities to import/export UML models that support the smooth integration with UML modelling tools. Besides, the designer can express the transformations in a declarative style that eases the development of the barb. VIATRA is integrated in Eclipse, and using Rational Software Architect for UML, which is also an Eclipse plug-in, we get a uniform development environment. Finally, using the Eclipse based SENSORIA case tool, we can deploy the verification tool as a service, in the same environment.

The second, conceptually more important, issue with respect to BMDSD relates to the reciprocal influence of the spine and the barbs in terms of modeling concepts and development process architecture, i.e. what to model, how, and when. The case we deal with, Politically Correct, is exemplar in this respect. $\lambda^{\mathtt{req}}$, the underlying theory [7], entails a very specific approach to the specification and execution of service based applications: the interface of a service comprises an abstraction of its behaviour, and the execution model carries on an analysis of these abstractions to select the proper services to invoke at run–time. One of the goals of our work is to understand how much this approach may be compatible with more traditional SOA settings.

The cognitive burden of the developer is also an issue. How many conceptual frameworks does he need to master, and how many related notations, to carry on his work proficiently, in a barbed development? We think that UML may help in providing a unifying framework for the different barbs, thanks also to its ability to support different views of the models under development, and to integrate various modelling concerns thanks to its *stereotype* extension mechanism. The obvious way to lower the cognitive burden is to distribute it among several developers, each with his own expertise. However, the issue of keeping the model needed for the various concerns consistent, still remains, and we want to assess how much the aforementioned characteristics of UML can help in facilitating the development of consistent models.

3 Technologies for BMDSD

3.1 UML profiles

Profiles give the ability to tailor the UML language for different application areas, while maintaining interoperability among tools [24]. Indeed, tools are defined on the standard meta–model, which is not changed by the definition of a new profile: only some limited additions are permitted. The profiles mechanism is consistent with the OMG Meta Object Facility (MOF).

A profile is a package that defines a subset of the UML meta–model elements, and defines constraints, stereotypes and tags to be applied to this set. A profile can be used in a UML model by applying it.

A stereotype permits to extend the UML language by adding new elements. This is done, by adding a new class in the meta–model, as a specialization of an existing one. Tags are used to add properties to a UML element by adding new attributes to the corresponding meta–model class. A constraint specifies a restriction of the meta–model element it is applied to. It can be written in natural language or in the UMLs Object Constraint Language (OCL).

3.2 Model transformation in VIATRA

VIATRA (VIsual Automated model TRAnsformations) is a framework for model transformation [6] [4]. It includes an engine to derive the target model fully automatically.

VIATRA focuses on *precise* model–driven development: the model transformations are specified in a mathematically precise way. A model transformation takes as input a model conforming to a given meta–model and produces as output another model conforming to the same or to a different meta–model.

Meta–models and models are described in VIATRA using the VIATRA Textual Modeling Language (VTML), within the Visual and Precise meta–modeling (VPM) approach [28]. The basic elements of a meta–model in VIATRA, are entities –a generalization of the Meta Object Facility (MOF) package, class, and object concepts– and relations – a generalization of MOF association end, attribute, link end, and slot concepts. Relations may have an associated multiplicity and a generalization relation may be defined between entities and/or relations.

[4] The current version of the framework is called VIATRA2. However, we simply call it VIATRA.

```
//***************** POLICY ENTITIES ******************//
entity(Policy);
entity(State);
entity(InitState);
entity(FinalState); // the error state in policies
entity(Symbol);

entity(Transition);
entity(Guard);
entity(Effect);

//***************** POLICY RELATIONS *****************//
relation(name, Policy, String);
relation(initstate, Policy, InitState);

relation(name, State, String);

relation(source, Transition, State);
relation(target, Transition, State);
relation(name, Transition, Symbol);
relation(guard, Transition, Guard);
relation(effect, Transition, Effect);

relation(representation, Symbol, String);

//**************** POLICY DEPENDENCIES *****************//
supertypeOf(State, InitState);
supertypeOf(State, FinalState);
```

Table 1
The policy meta--model

As an example we show in Table 1 part of the meta–model we use in the case study, relative to the representation of policies as automata. Its interpretation should be clear, taking in mind that the first argument of a (binary) relation is its name, and the other two are the related entities. A model conforming the given meta–model is shown in Figure 6, and will be discussed later.

Model transformations are defined by precise rules in two mathematical formalisms, graph transformation (GT) and Abstract State Machines (ASM). GT transformations are specified by pre– and post–condition patterns, and an *action* part which defines additional side-effects of a rule. ASMs provide for complex model transformations with all necessary control structures of imperative programming languages. An ASM machine is defined by a set of rules in the form pattern–action and functions.

Even though the VTML language, the VPM approach, and the transformation formalisms are nonstandard, VIATRA provides importing and exporting tools from and to other formalisms. Among them, VIATRA provides an importer of UML2.0 models of IBM Rational Software Architect software, version 6.0. Moreover, VIATRA is an Eclipse plugin, and imports Eclipse Modeling Framework (EMF) models using an appropriate importer plugin. Indeed, among the goals of the VIATRA project, there is the on–line synchronization between EMF and VPM models.

In particular, with respect to UML, VIATRA includes a facility to import UML models abiding the EMF UML2 meta–model, and therefore supports their transformation. Facilities to parse extended meta–models (profiles) are under development.

3.3 The SENSORIA Case Tool: XML and EMF

Within the SENSORIA project a Case Tool is being developed, to provide a platform for the integration of the SENSORIA tools. The platform architecture is service–oriented: tools are published as services, they can be discovered, used, and orchestrated. The SENSORIA Case Tool itself is built as a SOA, and implementation is being done on top of the (Equinox implementation of the) OSGi platform. In our case, we deal with the integration of three tools, namely Software Architect, VIATRA, and Politically Correct.

In the OSGi platform, the exchange of data between services is based on Java objects, which should instantiate a meta model in EMF. This is not a strict constraint for the SENSORIA tools, since any XML based representation of data will serve. However, the EMF allows one to directly use EMF based transformation tools like VIATRA.

4 Case study: Politically Correct in UML

To experiment with a barb in the advocated development approach, we are designing and implementing *Politically Correct in UML* (PCU), a service for the verification of service based software systems. PCU is intended to wrap a UML interface around Politically Correct (PC) [19], one of the verification tools being developed in SENSORIA as part of the work on $\lambda^{\mathtt{req}}$ [7].

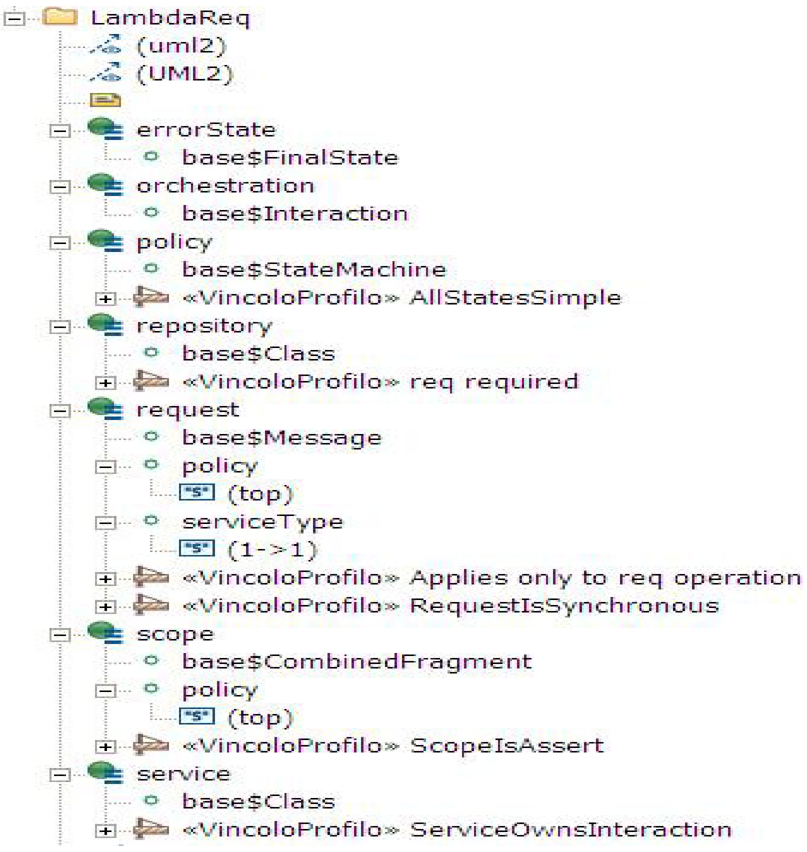
The $\lambda^{\mathtt{req}}$ theory underlying the specification and verification approach of PC is based on a few tenets:

- a service interface comprises a signature and a *history expression* that over–approximates its *public* behavior, in terms of all the possible traces (*histories*) of *visible* actions;
- these interfaces are available to service developers in a repository, and they are certified, i.e. their implementations abide to their specifications;
- the approach, with the typical style of SOA, allows developers to define new services by orchestrating other services by *call by contract*, a mode of call that includes constraints on the requested interfaces. Constraints are called *policies*, and are defined as automata: a history expression satisfies a policy if all the traces it defines are accepted by the policy automaton;
- policies can also be used to constrain a piece of orchestration, enclosing it in a *scope* with an associated policy.

The original target of $\lambda^{\mathtt{req}}$ was security: in fact it can be used to express more general constraints, since it all depends on what is considered *visible*, and used in the interfaces and policies.

4.1 The UML interface for Politically Correct

We assume the spine model development to be carried on in UML: the designer specifies services and policies in UML, extended with the $\lambda^{\mathtt{req}}$ profile, which we



```
LambdaReq
    (uml2)
    (UML2)

    errorState
        base$FinalState
    orchestration
        base$Interaction
    policy
        base$StateMachine
        «VincoloProfilo» AllStatesSimple
    repository
        base$Class
        «VincoloProfilo» req required
    request
        base$Message
        policy
            (top)
        serviceType
            (1->1)
        «VincoloProfilo» Applies only to req operation
        «VincoloProfilo» RequestIsSynchronous
    scope
        base$CombinedFragment
        policy
            (top)
        «VincoloProfilo» ScopeIsAssert
    service
        base$Class
        «VincoloProfilo» ServiceOwnsInteraction
```

Fig. 1. The $\lambda^{\texttt{req}}$ profile

will discuss in a while. In particular, the designer specifies the relevant dynamic facets of each service, i.e. the services it requires and those actions of its that are relevant to the constraining policies. This behavior is specified by a UML interaction owned by the service. These two elements –signature and interaction–, together with the policies, are what $\lambda^{\texttt{req}}$ requires to specify a service: from such a PCU model, we can extract the representation that PC needs to perform its analysis. Policies are specified using UML state machines. Indeed, policies are defined in $\lambda^{\texttt{req}}$ by suitable automata [8] that over–approximate the admissible execution traces of the services. These automata can be recovered from the profiled representation as State Machines, introduced by the $\lambda^{\texttt{req}}$ profile.

Figure 1 presents the $\lambda^{\texttt{req}}$ profile, as it appears in the RSA Model Navigator. The profile introduces six stereotypes. The base class of stereotype B appears as an attribute, with name `base$B` in Figure 1. The tags values are also listed as attributes, with their default value listed below, when defined. For policies, the value `top` references the most permissive policy, i.e. the one that excludes no behavior. The constraints defined in the profile are stereotyped VincoloProfilo[5] in RSA. For the sake of space, only the name of each constraint is shown: still, it

[5] Italian for Profile Constraint.

should be enough to convey at least the purpose of the constraint.

In this PCU case study, we could quite naturally exploit standard UML concepts (as base meta–classes for the new stereotypes) to capture new advanced modelling concepts. As already said, this has the consequence that standard tools can be easily adapted and offered to the designers. At the same time, the new concepts are framed in a well known context, hopefully accelerating their adoption. We expect to assess the acceptance of the profile for PCU, by interacting with industrial partners.

The next subsections outline how to transform the UML model into a suitable input for PC.

4.2 Model transformation

Despite the just mentioned advantages, a UML model usually has a major defect, when exported from a tool like RSA: it is a huge beast, since it has to cater for the many concerns of the development process. Instead, the analysis tools usually work on models built on a small set of concepts, with ad hoc notations. Hence the need of extracting from the UML model the input for the analysis tool, nowadays usually some form of XML presentation. This could be done in one step, directly transforming the UML models into some linearized form that the relevant analysis tools can accept. This was done in some previous work, but lacks of flexibility and open-ness. The approach we are following, which is also coherent with the SENSORIA main stream with respect to model representation, is to pass from the UML model to another model, which abides to a suitably defined, as simple as possible meta-model. This makes almost straightforward the linearization of the target model for tool input, and also paves the way to other utilizations of the target model by transformation tools. The SENSORIA context suggests the adoption of EMF as the modelling framework for PCU.

The target meta-model for PCU has two major components. The first one is a straightforward representation of the abstract syntax of the $\lambda^{\mathtt{req}}$ expressions: the corresponding models are extracted from the interactions, i.e. from the sequence diagrams. The second part caters with the policies, i.e. these models are extracted from the state machines: we adapted a standard notation for representing automata, [5].

The extracted models can be exported as such, i.e. as EMF files, or linearized in XML according to a suitable XSD, for successive input to other tools. In our case, a suitable interface to Politically Correct is being defined.

4.3 VIATRA transformation: discussion

VIATRA combines graph transformations and abstract state machines into a single approach, and proved to be a powerful model transformation tool. Among the main advantages of using this framework instead of using a standard programming language, we note that rule patterns make it easier to adapt a transformation specification in the case of changes of the source or target metamodels. This has proved to be a useful feature, especially when dealing with verification tools at their early stages of development. We expect that the declarative specification style that will be provided by the next version of VIATRA will make the work even smoother.

Moreover, at least on the small models we experimented with, performance is not an issue.

Due to the lack of full handling of profiles in the available VIATRA version, PCU currently rather than dealing with a single model comprising both interactions and policies deals with two separate models that are extracted from the comprehensive one. The next version of VIATRA should solve this problem.

5 A concrete scenario: car repair

To experiment with the PCU barb, we exploit one of the SENSORIA case-studies, the Car repair scenario [16]. The PCU specification presented here follows the one given in $\lambda^{\texttt{req}}$ in [10].

Assume a car equipped with a diagnostic hardware/software system that continuously reports on the status of the vehicle. When the car experiences some major failure (e.g. engine overheating, exhausted battery, flat tires) an embedded service is invoked to provide the user with a tow-truck that carries his damaged car to a garage for repair. We are interested in a SOA based design and implementation such that the selection may take into account some driver personalized policies, e.g. avoiding garages that did not service well in the past, as well as other global constraints, e.g. the truck should be close enough to both the damaged car and the garage. The design naturally leads to identify the need of three kinds of services: The *CarEmergency* service, which is invoked by the diagnosis subsystem, and looks for tow-trucking and repair; the *TowTruck* service, which manages towing, and the *Garage* service that manages car repair.

5.1 Car repair in PCU

As usual in UML, we consider separately static and dynamic modelling. The static facets are given in the class diagram of Figure 2.

The *CarEmergency* service is invoked by the embedded diagnosis system, each time a fault is reported. The actual kind of fault, and the location of the car, are passed as arguments: the signature of the service is exposed by the public standard operation `main` of the service, i.e. we require the designer to define the service signature through this operation[6].

For instance, the sequence diagram in Figure 3 specifies that *CarEmergency* needs two services and may perform the visible action `blacklist`. Consider first the pair of requests: the first one looks for the tow-truck service, and the second one for the garage service[7]. The contract policy withinReach(location) requires

[6] In Figure 2 we have annotated the diagram with the signature of the operations. This information is in RSA model, but cannot be shown in the class diagram, and is only visible in the Model Explorer pane.

[7] For simplicity, we assume that both the truck and the garage will be available. As for now, our UML profile and the related transformation tool need to be extended to cope with the $\lambda^{\texttt{req}}$ *planning block* notion

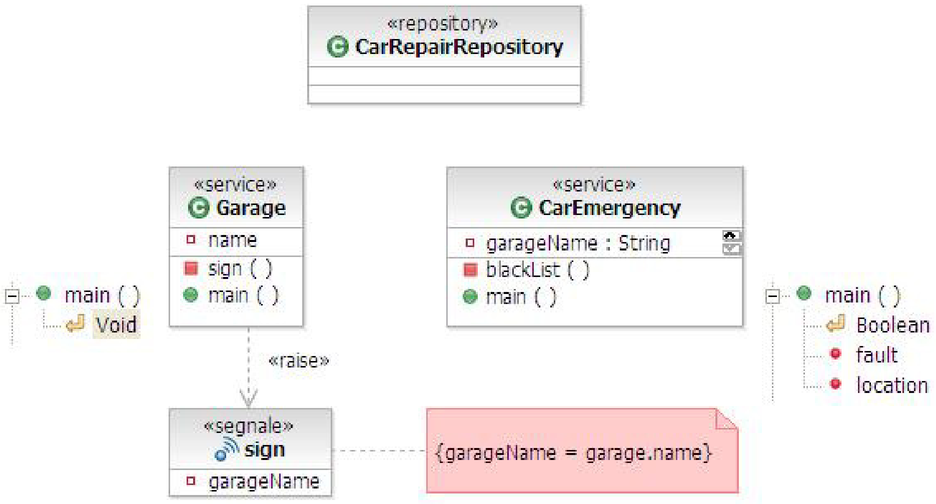


Fig. 2. Some involved services

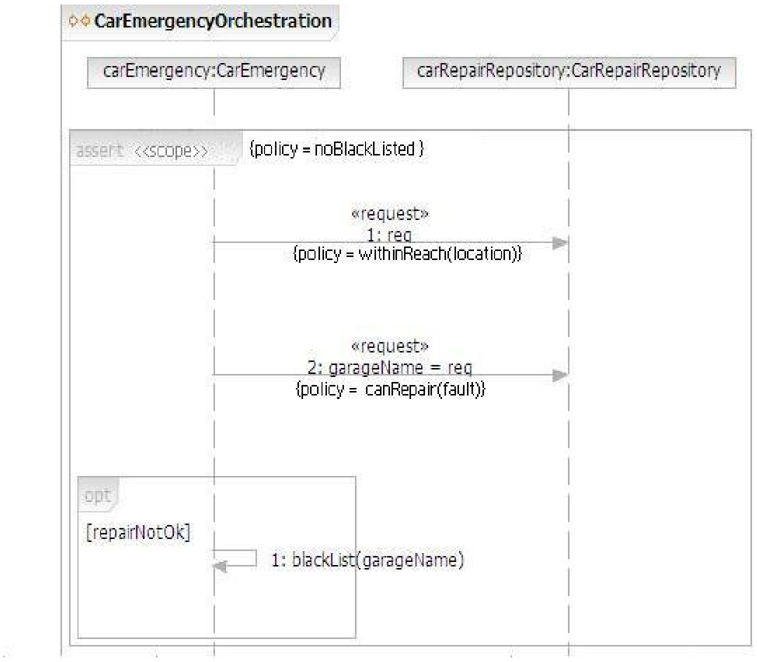

CarEmergencyOrchestration
carEmergency:CarEmergency
carRepairRepository:CarRepairRepository
assert <<scope>>
{policy = noBlackListed }
«request»
1: req
{policy = withinReach(location)}
«request»
2: garageName = req
{policy = canRepair(fault)}
opt
[repairNotOk]
1: blackList(garageName)

Fig. 3. The CarRepair interaction

that the tow-truck is able to serve the location where the customer is waiting for towing. The contract policy canRepair(fault) requires that the garage can repair the fault. In order to be candidate for invocation, each garage service exposes the faults it can handle, in the initial fragment of its own sequence diagram: therefore

that caters with the required transactional behavior.

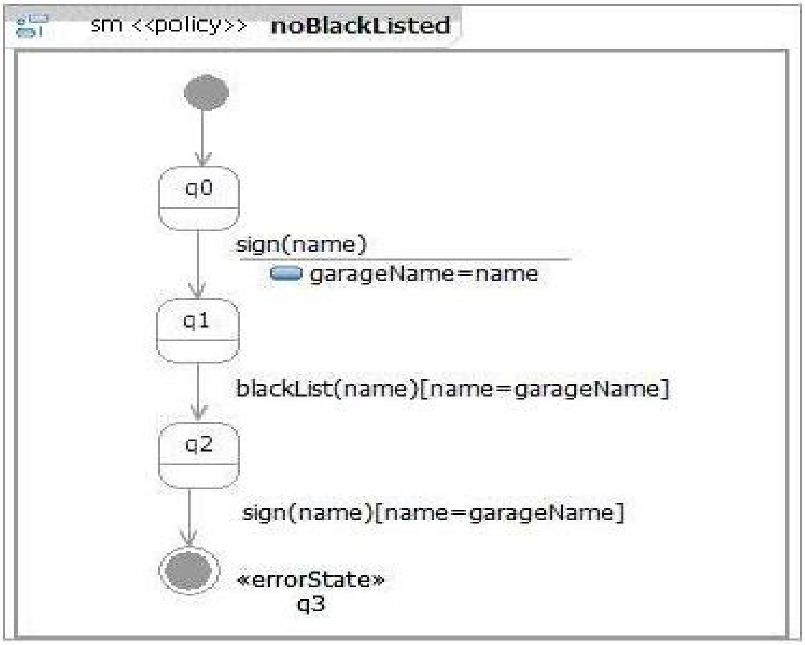

sm <<policy>> noBlackListed
q0
sign(name)
garageName=name
q1
blackList(name)[name=garageName]
q2
sign(name)[name=garageName]
«errorState»
q3

Fig. 4. An example policy: noBlackListed

they become part of the service interface, and can be matched with the caller requirements[8]. Going back to *CarRepair*, once the repair is over, the customer can assess the quality of the service he received from the garage, and possibly put the garage on a black-list. This list is exploited by the enclosing `assert` fragment, which introduces the policy that a black-listed garage cannot be selected for future emergencies, and requires that it is respected by all the enclosing behaviors. This policy, noBlackListed is shown in Figure 4 as a stereotyped state machine.

To understand the policy in Figure 4, we need to know that the sign(name) event is issued by a garage after a repair: therefore any history including the sequence sign(name), blackList, i.e. any history where a garage issues a receipt once black-listed, leads to an error state[9]. It is precisely this kind of policy breaches that the verification machinery of PC is able to detect. Assume as an example an instance of the scenario with a car CAR, two tow–track services T_1 and T_2, and two garage services G_{FL} and G_{LU}, where:

T_1 can serve Florence and Pisa;

T_2 can serve Pisa, Siena, and Lucca;

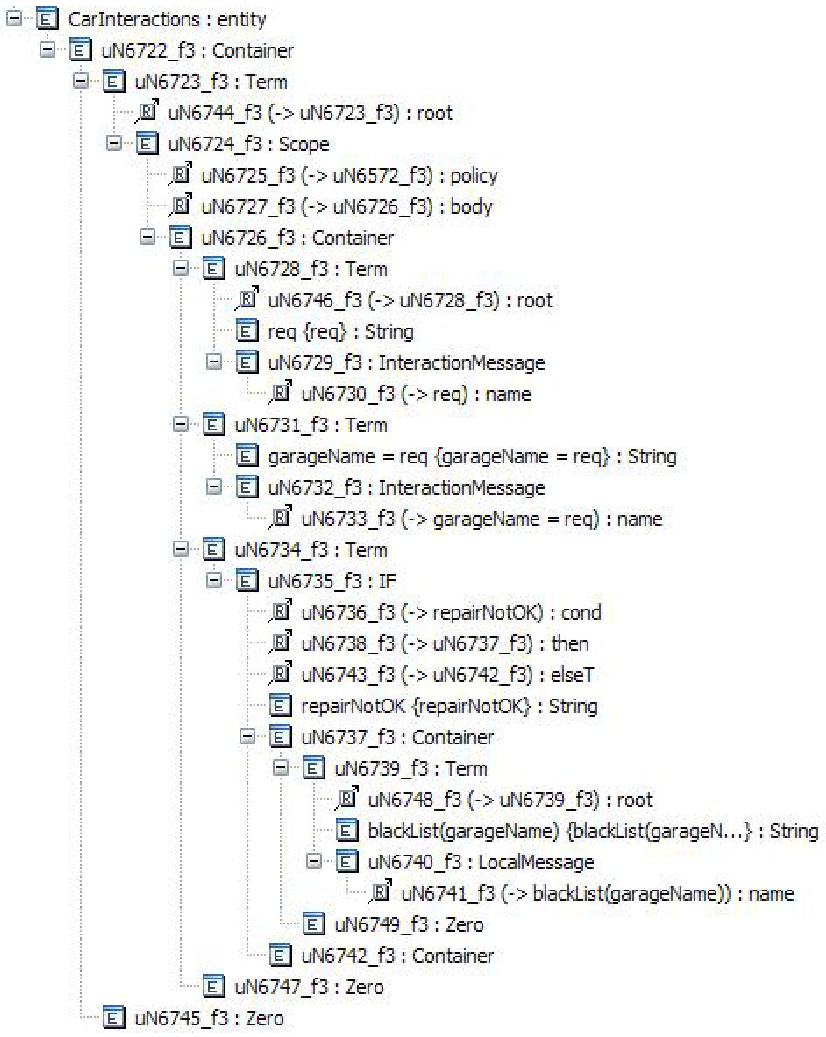
G_{FL} is a garage in Florence repairing tires and batteries;

G_{LU} is a garage in Lucca repairing engines and tires;

CAR is in Pisa with a flat tire, and it has black listed the garage in Lucca.

[8] In PC, it is assumed that when a service is published in the repository, its interface is formally certified.

[9] It should be pointed out that, though this description lends itself a run–time flavour, the analysis is performed prior to service invocation, during service selection.

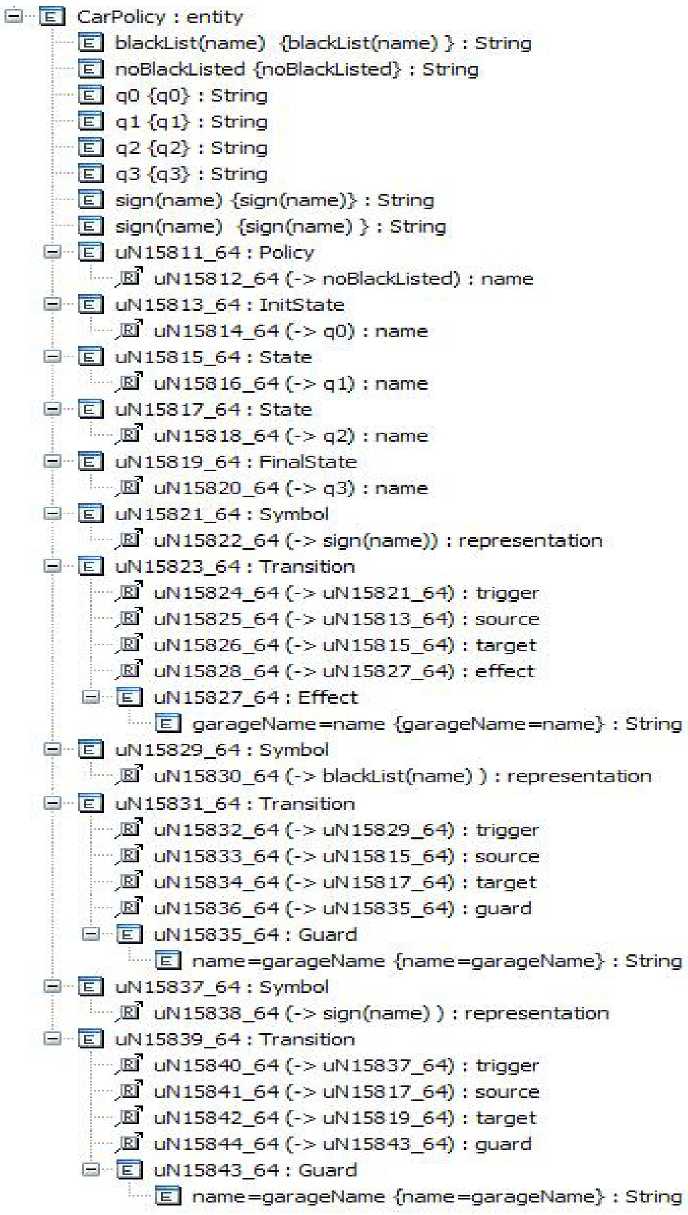


```
CarInteractions : entity
  uN6722_f3 : Container
    uN6723_f3 : Term
      uN6744_f3 (-> uN6723_f3) : root
      uN6724_f3 : Scope
        uN6725_f3 (-> uN6572_f3) : policy
        uN6727_f3 (-> uN6726_f3) : body
        uN6726_f3 : Container
          uN6728_f3 : Term
            uN6746_f3 (-> uN6728_f3) : root
            req {req} : String
            uN6729_f3 : InteractionMessage
              uN6730_f3 (-> req) : name
          uN6731_f3 : Term
            garageName = req {garageName = req} : String
            uN6732_f3 : InteractionMessage
              uN6733_f3 (-> garageName = req) : name
          uN6734_f3 : Term
            uN6735_f3 : IF
              uN6736_f3 (-> repairNotOK) : cond
              uN6738_f3 (-> uN6737_f3) : then
              uN6743_f3 (-> uN6742_f3) : elseT
              repairNotOK {repairNotOK} : String
              uN6737_f3 : Container
                uN6739_f3 : Term
                  uN6748_f3 (-> uN6739_f3) : root
                  blackList(garageName) {blackList(garageN...} : String
                  uN6740_f3 : LocalMessage
                    uN6741_f3 (-> blackList(garageName)) : name
                uN6749_f3 : Zero
              uN6742_f3 : Container
          uN6747_f3 : Zero
    uN6745_f3 : Zero
```

Fig. 5. EMF model of the interaction

Most pairs ⟨garage, tow–track⟩ violate some policies. For instance, $\langle G_{FL}, T_2\rangle$ is not a possible solution, since the tow–track does not reach Florence, and the garage in Lucca cannot be selected since it is blacklisted. Politically Correct is able to perform this analysis and to single out the only orchestration of services that does not violate any policy, namely $\langle G_{FL}, T_1\rangle$.

For completeness, the results of applying the transformation for the scenario above is shown in Figure 5 and 6, as they appear in the VIATRA VPM modelspace pane.



```
CarPolicy : entity
    blackList(name)  {blackList(name) } : String
    noBlackListed {noBlackListed} : String
    q0 {q0} : String
    q1 {q1} : String
    q2 {q2} : String
    q3 {q3} : String
    sign(name) {sign(name)} : String
    sign(name)  {sign(name) } : String
    uN15811_64 : Policy
        uN15812_64 (-> noBlackListed) : name
    uN15813_64 : InitState
        uN15814_64 (-> q0) : name
    uN15815_64 : State
        uN15816_64 (-> q1) : name
    uN15817_64 : State
        uN15818_64 (-> q2) : name
    uN15819_64 : FinalState
        uN15820_64 (-> q3) : name
    uN15821_64 : Symbol
        uN15822_64 (-> sign(name)) : representation
    uN15823_64 : Transition
        uN15824_64 (-> uN15821_64) : trigger
        uN15825_64 (-> uN15813_64) : source
        uN15826_64 (-> uN15815_64) : target
        uN15828_64 (-> uN15827_64) : effect
        uN15827_64 : Effect
            garageName=name {garageName=name} : String
    uN15829_64 : Symbol
        uN15830_64 (-> blackList(name) ) : representation
    uN15831_64 : Transition
        uN15832_64 (-> uN15829_64) : trigger
        uN15833_64 (-> uN15815_64) : source
        uN15834_64 (-> uN15817_64) : target
        uN15836_64 (-> uN15835_64) : guard
        uN15835_64 : Guard
            name=garageName {name=garageName} : String
    uN15837_64 : Symbol
        uN15838_64 (-> sign(name) ) : representation
    uN15839_64 : Transition
        uN15840_64 (-> uN15837_64) : trigger
        uN15841_64 (-> uN15817_64) : source
        uN15842_64 (-> uN15819_64) : target
        uN15844_64 (-> uN15843_64) : guard
        uN15843_64 : Guard
            name=garageName {name=garageName} : String
```

Fig. 6. EMF model of the policy

6 Related work

Model-Centric Software Development (MCSD) [29] is a form of model driven engineering in which models are central to all phases of the development process. The approach uses domain-specific modeling languages (to represent aspects of interest such as atomicity of data access, end-to-end message delay, and resource contention), automated generation of partial implementation artifacts (they can include fine-grained concrete functionality and specifications for software simulators and emulators), and model verification and checking as well as rapid-prototype generation. MCDS is an instance of BMDSD, but apparently the supporting environment is built in a rather ad hoc fashion.

Within previous EU projects, DEGAS and AGILE, several prototypical environments have been developed, well in line with BMDSD. *Choreographer* is a software tool platform which facilitates security and performance analysis of systems which starts and ends with UML model descriptions [14]. A UML project is presented to the platform for analysis, formal content is extracted in the form of process calculi descriptions, analysed with the analysers of the calculi, and the results of the analysis are reflected back into a modified version of the input UML model. The methodology integrates the use of ForLysa, a tool for security analysis [13] based on the LySa approach [12], and the PEPAWorkbench [3], based on PEPA [17]. The goals are very similar to ours, but the model transformations are built with ad hoc Java programs: we expect to benefit from the flexibility gained by using VIATRA in our experiments in generalizing the approach toward BMDSD.

As for the present work, the Choreographer platform is centered on a UML profile, called UMLsec [22]. This profile expresses security–relevant information within the diagrams in a system specification, and on the related approach to secure system development [20]. UMLsec includes elements to express security requirements, like fair exchange, secrecy/confidentiality, secure information flow, secure communication link. Validation is based on a formal semantics for the used fragment of UML, and on a formal notion of adversary.

Other related work include model transformation from protocols specified with sequence diagrams to first order logic theories [20], where verification exploits standard theorem-provers, like e-SETHEO, to reveal potential attacks: [21].

A further approach permits the verification of web services including asynchronous events, modelled as a set of communicating state machines. The UML model is transformed into a doubly labelled transition system, and then verified –by on-the-fly model checking– with respect properties in the UCTL temporal logic [27]. In this case, both model transformation and verification are supported by a unique tool, called UMC.

7 Conclusion

Barbed Model–Driven Software Development is a Model Driven Engineering approach where, besides a spine of model transformations moving from the most abstract, domain related models, to the most concrete, platform related models, there are a number of spikes, relating models in the spine to specialized models. These permit specific, sometimes very sophisticated, analysis of part of the software.

In this paper we report on ongoing work addressing experimentally the issues related to BMDSD, to get acquainted with the approach and the tools supporting the integrations of the tools used in the barbs. We describe the design and implementation of a specific barb, using specific technologies: the profiling capabilities of UML2 as supported by Rational Software Architect, the transformation framework

VIATRA, and the SENSORIA CaseTool. The case study is centered around a tool, named Politically Correct, that statically checks that an orchestration of servicies abides given policies.

The experimentation with VIATRA has been so far promising, both with respect to usability of the tool and performance. Notwithstanding, we plan to consider other transformation frameworks, still based on EMF, like Tiger [15] –currently also under development in SENSORIA–.

An emerging pattern from our experience is that the transformations that go from the spine to the other end of the barb look like *homomorphic projections* of one complex model in a simpler one: we plan to investigate this intuition and see if it may lead to a general framework to simplify the construction of the barbs on one side, and support the formalization of the BMDSD approach, e.g. in an algebraic setting [25,23], on the other. Besides, an algebraic formalization may also help in structuring the approach to the backward transformations, from the analysis results to the UML model. This is especially needed from a pragmatical point of view, when the analysis finds some errors. Preliminary results, rather ad hoc, are available, e.g. in [14], but a general theory of error reflection is still missing.

References

[1] Agile home page. http://www.pst.informatik.uni-muenchen.de/projekte/agile/.

[2] Degas home page. http://www.omnys.it/degas/.

[3] Pepa home page. http://www.dcs.ed.ac.uk/pepa/.

[4] Sensoria home page. http://www.sensoria--ist.eu.

[5] Y. An and S. Yu. Automl: Definition and implementation. Technical Report 629, Department of Computer Science, University of Western Ontario, Jan 2004.

[6] A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287. ACM Press, 2006.

[7] M. Bartoletti, P. Degano, and G.L. Ferrari. Planning and verifying service composition. Technical Report TR-07-02, Dipartimento di Informatica, Universitá di Pisa, 2007. To appear in Journal of Computer Security. http://compass2.di.unipi.it/TR/Files/TR-07-02.pdf.gz.

[8] M. Bartoletti, P. Degano, G.L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In H. Seidl, editor, *Proc. Foundations of Software Science and Computation Structures (FOSSACS'07)*, volume 4423 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2007.

[9] M. Bartoletti, P.Degano, and G.L. Ferrari. Security issues in service composition. In *Proc. FMOODS06*, volume 4037 of *Lecture Notes in Computer Science*. Springer, 2006. Invited talk.

[10] M. Bartoletti, P.Degano, G.L. Ferrari, and R. Zunino. Secure service composition. In A. Aldini, editor, *Proc. International Summer School on Security Analysis and Design*, Lecture Notes in Computer Science. Springer, 2006. To appear.

[11] H. Baumeister et al. D7.4a: Case tool: Initial requirements. SENSORIA project deliverable http://www.pst.ifi.lmu.de/projekte/Sensoria/del_12/D7.4.a.pdf, Aug 2006.

[12] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *16th Computer Security Foundations Workshop (CSFW 2003)*, pages 126–140. IEEE Computer Society Press, 2003.

[13] M. Buchholtz, C. Montangero, L. Perrone, and S. Semprini. For-LySa: UML for authentication analysis. In *2nd Int. Workshop on Global Computing (GC04)*, volume 3267 of *LNCS*, pages 92–105, Rovereto, Italy, 2004. Springer-Verlag.

[14] M. Buchholtz, S.Gilmore, V. Haenel, and C. Montangero. End-to-end integrated security and performance analysis on the DEGAS Choreographer platform. In I.J. Hayes J.S. Fitzgerald and A. Tarlecki, editors, *Proceedings of the International Symposium of Formal Methods Europe (FM 2005)*, number 3582 in LNCS, pages 286–301. Springer-Verlag, June 2005.

[15] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of visual editors as eclipse plug-ins. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, New York, NY, USA, 2005. ACM Press.

[16] S. Gnesi, N. Koch, and A. Zobel et al. D8.0: Case studies scenario description (automotive case study). SENSORIA project deliverable, 2006.

[17] J. Hillston. Process algebras for quantitative analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 239–248, Chicago, June 2005. IEEE Computer Society Press.

[18] IBM. Rational software architect. http://www-306.ibm.com/software/awdtools/architect/swarchitect/.

[19] E. Italiano. Politically correct. Master's thesis, University of Pisa, 2006. In Italian. http://etd.adm.unipi.it/theses/available/etd-03062007-123926/.

[20] J. Jürjens. *Secure Systems Development with UML*. Springer–Verlag, 2004.

[21] J. Jürjens and T.A. Kuhn. Automated theorem proving for cryptographic protocols with automatic attack generation. Technical Report TUM-I0424, Institut für Informatik, Technische Universität München, 2004.

[22] Jan Jürjens. Umlsec: Extending uml for secure systems development. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425, London, UK, 2002. Springer-Verlag.

[23] C. Montangero. Program specification and optimization of functions over recursive data structures. In *3rd International Symposium on Programming*, pages 336–351, Paris, 1978.

[24] OMG. Uml superstructure specification, version 2.1.1, formal/07-02-05. Download from http://www.omg.org/technology/documents/formal/uml.htm, Feb 2007.

[25] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[26] D.C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, Feb 2006.

[27] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of an aynchronous protocol for service-oriented applications. In *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, LNCS, Berlin, Germany, July 1–2 2007. Springer-Verlag. To appear.

[28] D. Varró and A. Pataricza. Vpm: A visual, precise and multilevel metamodeling framework for describing mathematical domains and uml. *Journal of Software and Systems Modeling*, 2(3):187–210, 2003.

[29] D. Waddington and P. Lardieri. Model–centric software development. Sidebar to [26].