Abstraction-raising Transformation for Generating Analysis Models

Antonino Sabetta[1], Dorina C. Petriu[2], Vincenzo Grassi[1], Raffaela Mirandola[1],

[1] University of "Tor Vergata", Dept. of Informatics, Systems and Production
Rome, Italy
{sabetta,vgrassi,mirandola}@info.uniroma2.it
[2] Carleton University, Department of Systems and Computer Engineering
Ottawa, ON Canada, K1S 5B6
petriu@sce.carleton.ca

Abstract. The verification of non-functional requirements of software models (such as performance, reliability, scalability, security, etc.) requires the transformation of UML models into different analysis models such as Petri nets, queueing networks, formal logic, etc., which represent the system at a higher level of abstraction. The paper proposes a new "abstraction-raising" transformation approach for generating analysis models from UML models. In general, such transformations must bridge a large semantic gap between the source and the target model. The proposed approach is illustrated by a transformation from UML to Klaper (Kernel LAnguage for PErformance and Reliability analysis of component-based systems).

1 Introduction

OMG's Model Driven Architecture (MDA) promotes the idea that software development should be based on models throughout the entire software lifecycle [13]. This change of focus from code to models raises the need for formal verification of functional and non-functional characteristics of UML software models. Over the years, many modeling formalisms (such as queueing networks, Petri nets, fault trees, formal logic, process algebras, etc.) and corresponding tools have been developed for the analysis of different non-functional characteristics (such as performance, reliability, scalability, security, etc.). The challenge is not to reinvent new analysis methods targeted to UML models, but to bridge the gap between UML-based software development tools and different existing analysis tools.

Each of these analysis models and tools is suited for the evaluation of different non-functional software properties. In general, an analysis model abstracts away many details of the original software model, emphasizing only the aspects of interests for the respective analysis. A transformation whereby a more abstract target analysis model is generated from a source software model is called here "abstraction-raising" transformation, as opposed to a "refining" transformation that produces a more detailed target model (such as the transformations used in MDA).

Traditionally, analysis models were built "by hand" by specialists in the field, then solved and evaluated separately. However, with the change of focus on models brought by MDA, a new trend started to emerge, whereby software models are automatically transformed into different analysis models. For example, this kind of approach was used to obtain a formal logic model for analyzing security characteristics in [7]. Transformations from UML into different performance models have been surveyed in [1]. Examples of such transformations are from UML to Layered Queueing Networks in [10, 11], to Stochastic Petri Nets in [2], and to Stochastic Process Algebra in [3]. More recently, a transformation framework from multiple design models into different performance models was proposed in [17].

Different kinds of analysis techniques may require additional annotations to the UML model to express, for instance non-functional requirements and characteristics, or the user's directives for the desired analysis. OMG's solution to this problem is to define standard UML profiles for different purposes. Two examples of such profiles are the “UML Profile for Schedulability, Performance, and Time”[14] and "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms"[15].

The paper proposes a new approach for developing abstraction-raising transformations from UML to different kind of analysis models in different formalisms (such as Petri nets, queueing networks, fault trees, formal logic, etc.) In general, such transformations must bridge a large semantic gap between the source and the target model, which represent the system at different abstraction levels. The proposed approach is illustrated by a transformation from UML to Klaper (Kernel LAnguage for PErformance and Reliability analysis of component-based systems) [6].

2. Conceptual Description of the Transformation Approach

The proposed approach combines two methods that, so far, have been used separately in model transformations: relational and graph grammar-based transformations [5]. In the relational approach, used in the proposal for the QVT standard [12], the source and target models are each described by its own metamodel; a transformation defines relations (mappings) between different element types of the source and target (meta)models. According to [12], a *Transformation* is a generalization of both *Relation* and *Mapping*. Relations are non-executable bi-directional transformation specifications, whereas Mappings are unidirectional transformation implementations used to generate a target model from the source model.

The graph-transformation and relational approaches are compared in [8]. While the former is based on *matching and replacement*, the latter is based on *matching and reconciliation*. The conclusion is that, is spite of their differences, advantages and disadvantages, the two approaches are rather similar. More research is needed to identify which one is more suitable for certain kinds of applications.

In our proposed approach, we keep the idea that the source and target models are described by separate metamodels, between which transformations must be defined. However, in our case the target metamodel represents analysis domain concepts, which are usually at a higher-level of abstraction than the source model concepts. In order to define mappings between the source and target models, sometimes it is necessary to group (aggregate) a large number of source model elements according to certain rules, and to map the whole group to a single target element. The aggregation rules correspond to the raising in the abstraction level necessary for bridging the semantic gap between the source and the target model. Such rules are dependent on the semantic differences between the source and target model, and are not represented in the source metamodel.

Therefore, a new mechanism is needed to express the aggregation rules, in addition to the mechanism for defining the transformation from source to target. We propose to use a graph grammar [16] for describing the aggregation rules; the terminals of the graph grammar correspond to metaobjects from the source model, whereas the non-terminals correspond to more abstract concepts that will be transformed directly in target model concepts. The proposed transformation approach is illustrated in Fig.1. Some target model elements can be obtained by a direct mapping from source models elements, like in a relational transformation, whereas other target elements, representing more abstract concepts, correspond to graph-grammar non-terminals obtained by parsing the source model. According to the taxonomy of model transformations proposed in [9], the abstraction-raising transformation discussed in this paper is both exogenous (i.e., the source and target models are different) and vertical (the abstraction level is different).

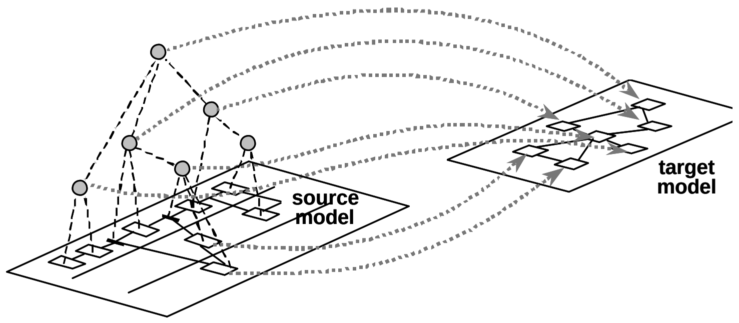


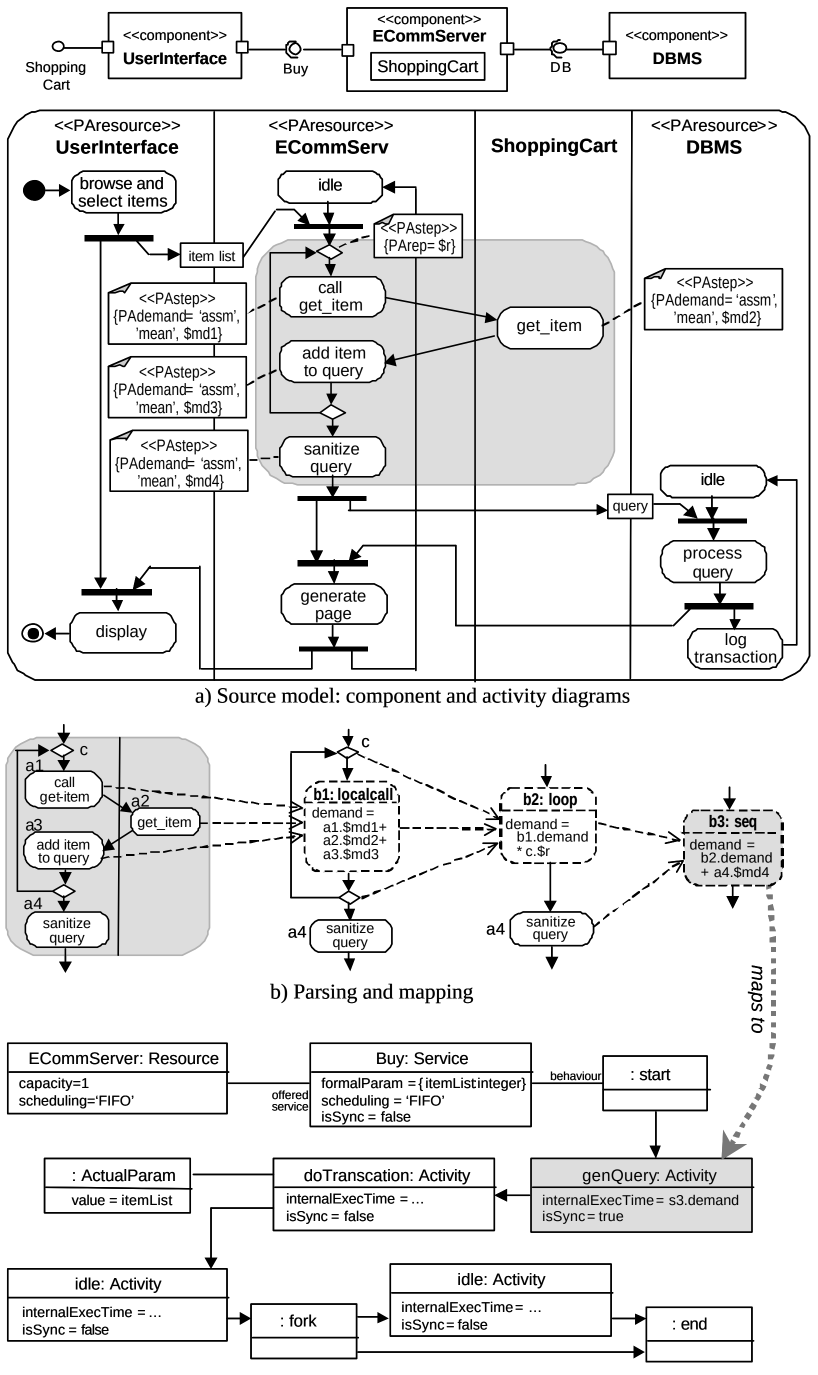
Figure 1. Principle of the proposed "abstraction-raising" transformation approach

3. Analysis Model for Component-based Systems

In this section, the abstraction-raising transformation approach presented in section 2 is illustrated by applying it to the transformation of a UML model extended with the SPT Profile to an analysis model named Klaper (Kernel LAnguage for PErformance and Reliability analysis of component-based systems) [6].

3.1. Description of the Source Model

We assume that the source UML model describes the high-level software architecture as a component diagram, and the key scenario(s) for performance/reliability analysis as activity diagram(s), as in Fig. 2.a. We also assume that the information required for the generation of the analysis model is available from the source UML model, possibly by means of annotations compliant with one or more profiles [14, 4].



a) Source model: component and activity diagrams

b) Parsing and mapping

c) Klaper model for the ECommServ component

Figure 2. Example of abstraction-raising transformation from UML to Klaper

Although both the parsing and the mapping implied by the proposed transformation are defined at the abstract syntax level based on the metamodel representation of the source model, the paper uses only the graphical UML notation for the sake of conciseness. The example is an e-commerce application designed as a client/server system with three basic components: a user interface, an e-commerce server component and a database (at the top of Fig. 2.a). Due to space limitations, we consider a single usage scenario, which is part of the checkout procedure, given as an activity diagram in Fig. 2.a. After the user has selected one or more items, the first client-server interaction takes place, with UserInterface acting as client of the ECommServer. We assume that a server component waits for requests in the "idle" state. After accepting a client request, ECommServer loops through the items in the shopping cart to prepare a database query, and acts in turn as client in another client-server interaction, where the DBMS is the server. After getting the required information, ECommServer generates a page with the checkout information, sends it to the UserInteface for display and returns to the "idle" state, where it can accept a new request.

The SPT Profile is used to identify the main basic abstractions for performance analysis in the UML model from Fig. 2.a. *Scenarios* define response paths through the system, and can have QoS requirements such as response times or throughputs. Each scenario is executed by a *workload*, which can be closed or open. Scenarios are composed of scenario *steps* that can be joined in sequence, loops, branches, fork/joins, etc. A step (stereotyped as <<PAstep>>) may be an elementary operation at the lowest level of granularity, or a complex sub-scenario. Each step has a mean number of repetitions, a host execution demand, other demand to resources and its own QoS characteristics, which are given as tagged value. *Resource* is another basic SPT abstraction; it can be active or passive, each with its own attributes.

Not all SPT annotations are shown in Fig. 2.a, just a few <<PAStep>> stereotypes applied to different activities. The tagged value PAdemand gives the CPU demand for a step. For instance, "PAdemand ='assm', 'mean', $md1, 'ms'" means that the step has an assumed mean execution time of $md1 ms (where $md1is a variable). The tagged value "PArep =$r" gives the average number of loop repetitions. Such quantitative annotations will be used during the transformation process to compute the parameters of the target model. A more detailed description of the use of SPT profile for performance analysis is given in [11].

3.2 Description of the Target Model

The target model in this case study is an abstract analysis model expressed in Klaper, a Kernel Language for Performance and Reliability analysis of component-based systems [6]. Its purpose is to capture in a lightweight and compact model all the relevant information for the performance and reliability analysis of component-based systems, while abstracting away irrelevant details. Klaper was designed as an intermediate "distilled" language to help bridge the large semantic gap between design-oriented and analysis-oriented notations, and to mitigate the "*N-by-M*" problem of translating *N* design notation types into *M* performance/reliability model types.

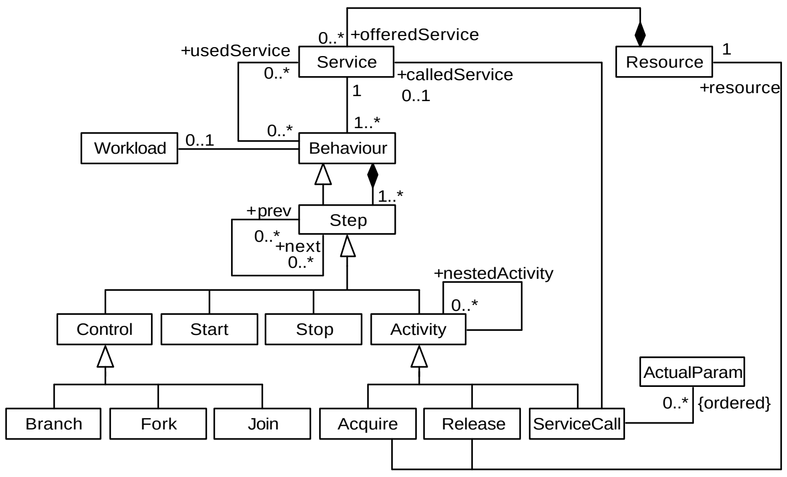


Figure 3. Klaper metamodel (adapted from [6])

This "*N-by-M*" problem is reduced to a less complex task of defining *N+M* transformations: *N* from different design notations to Klaper, and *M* from it to different analysis models. (In this paper, we consider only one transformation, from UML 2 to Klaper). Klaper has been defined in [6] as a MOF-compliant metamodel to allow the exploitation of existing transformation facilities in the context of MDA. A diagram of the metamodel is shown in Fig.3. The domain model underlying Klaper considers that a system is an assembly of interacting *Resources*, where a resource may offer (and possibly require) *Services*. Thus, Klaper Resources can represent both software components and physical resources like processors, communication links or other physical devices. Each offered Service is characterized by a list of *formal parameters* that can be instantiated with actual values by other resources requiring that service. The *Behaviour* of (offered) services is modeled as a graph of *Steps* that can be simple "internal" *Activities* (i.e. operations that do not require any services offered by other resources), or *Activities* with one or more associated *ServiceCalls* addressed to other Resources, or *Control nodes* (Begin/End, Fork/Join, etc.). An interesting feature of Klaper is that service parameters are meant to represent abstractions (for example expressed in terms of random variables) of the "real" service parameters (see [6] for more details).

3.3. Model Transformation

From a high level perspective, the mapping from UML to Klaper can be described as follows. UML components (from component diagrams) and nodes (from deployment diagrams) are mapped onto Klaper *resources*. The corresponding offered and required *services* are derived from the provided and required interfaces for each component. The *behaviour* of each offered service is derived from a suitable UML activity or state diagram, that either specifies the local component behaviour or the global system behaviour. Due to space constraints, we do not describe in this example the mapping of UML nodes to Klaper resources, nor the derivation of the behaviour that models the interactions between components (i.e. connectors). The attributes of Klaper elements defined in [6] are mainly derived from the information provided by the SPT [14] and reliability stereotypes [4] given in the UML source model.

Many of the mappings from UML to Klaper are straightforward, in that they can be described as one-to-one relations between elements of the two metamodels; for instance each UML *component* is mapped to a Klaper *resource*, each provided interface of a UML component is mapped to a *service* offered by the *resource* corresponding to that component, each required interface is mapped to a *service call*, and so on. However, there are cases where a group of elements in the source model represents, as a whole, an abstraction that will be mapped to a single Klaper element. To illustrate more clearly this idea, let us examine the derivation of the Klaper model for the ECommServ component (see Fig. 2.c).

In general, the behaviour is represented in the analysis model at a higher level of abstraction than in the source model; this comes from the nature of the transformation from a software design to a performance model. Thus, we do not need to translate each and every UML activity to a Klaper step, but would like to aggregate unnecessary details. For instance, we may decide that all the activities executed in a single swimlane between the receiving of a message and the sending of the next message (as shown in the shaded fragment in Fig. 2.a) should be grouped and mapped, as a whole, to a single Klaper *activity* (shown by the "maps-to" arrow in Fig. 2). We may also want to aggregate, in the same block, calls to local passive objects. The fragment shown in Fig. 2.a is rather simple, but in principle can have any number of activities connected in different ways in sequence, branches, loops, etc. Since the UML metamodel does not define a concept (metaclass) corresponding to a "block of activities" as described above, there is no single element in the source metamodel that can be mapped to an element in the target metamodel. We propose to describe the above aggregation rules by the means of a few graph grammar rules (see Fig. 4).

By applying the grammar rules in an appropriate order, we can eventually reduce a "correct" activity diagram to the starting symbol 'AD'. In the parsing process, a set of non-terminals are generated, which correspond to more "abstract" constructions found in the source model. The rules are applied for reduction as follows: when a subgraph matching the right hand side (rhs) of a rule is found in the host graph (i.e., in the source model possibly rewritten by previous rules), the matching subgraph is removed and is replaced by the left hand side (lhs) of the rule, according to the embedding policy. More precisely: a) the edges attached to nodes of the rhs that are kept in the lhs are preserved (they represent the "context"); b) the edges that are left dangling (because of the removal of some node from the lhs) are removed; c) if a node in the rhs is rewritten as one node in the lhs, then all the edges attached to the former are redirected to the latter (this applies to non-injective morphisms too). The graph grammar is structured so that high level constructs, such as loops, conditional constructs, sequences and client-server interactions, are discovered through parsing. To this purpose the concept of a "block" has been introduced and formally defined in the grammar by rule 2. Most of the rules are recursive, raising the abstraction power of the proposed technique.

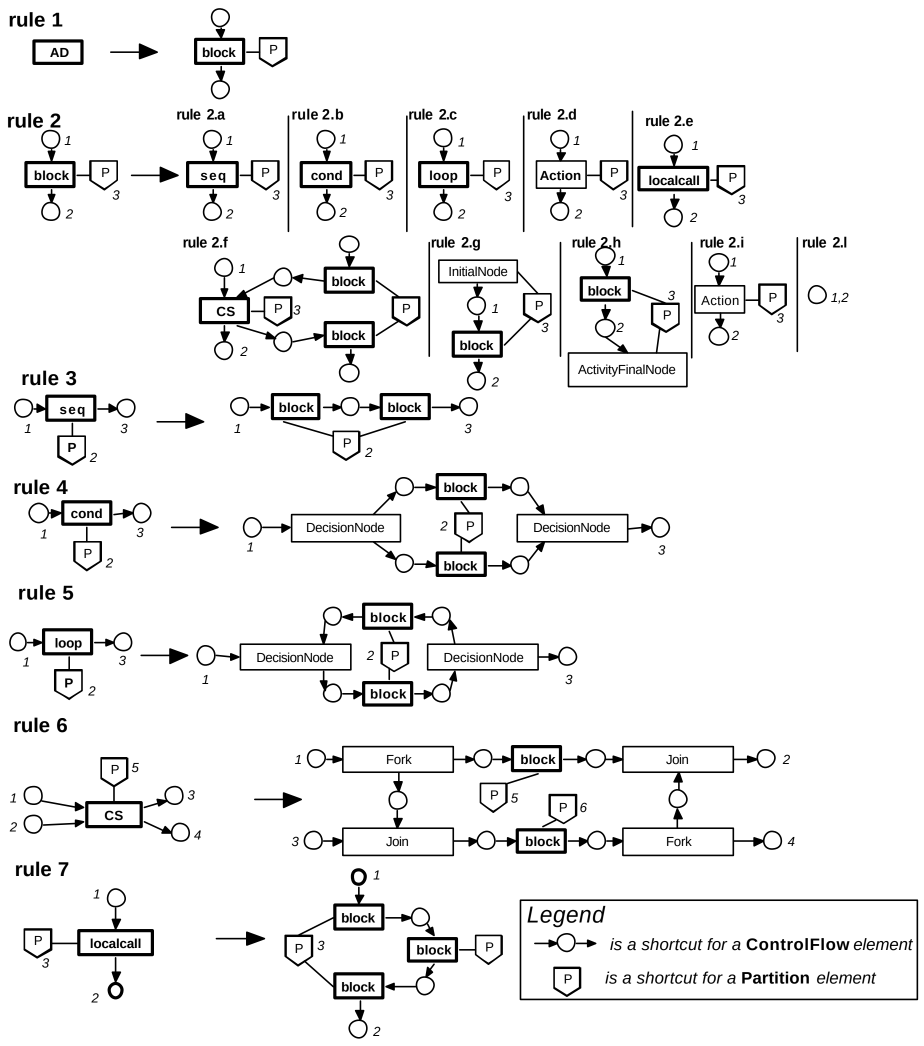


Figure 4. Graph grammar

Basic constructs such as sequences, conditional blocks and loops are defined, in terms of blocks and terminal symbols, by rules 3, 4 and 5. Rule 6 is used to recognize the asynchronous client-server interactions and rule 7 reduces calls to passive objects. Remarkably, each of the *blocks* in the right hand side of the grammar rules can represent structures as simple as an elementary action or as complex as a big block that could contain in turn other client-server interactions and arbitrarily nested conditions, loops and sequences.

Fig. 2.b illustrates how the rules are applied in order to aggregate the activities from the shaded fragment to a single block. In order to keep the figure clear, a few details of the transformation steps were omitted. In the first step, rule 2.d is applied to each of the actions to rewrite them as blocks. Then blocks a1, a2 and a3, are reduced by rule 7 to a *localcall* non-terminal (b1). Non-terminal elements have their own attributes, computed from the elements in the right-hand side of the reduction rule, possibly by considering also the stereotypes and tagged values attached to them. In this example, the attribute *demand* of *b1* (which represents the average CPU execution time required for this block) is computed as the sum of the mean execution times for the activities *a1 a2* and *a3* (given as SPT performance annotations in Fig.2.a). In the second step, rule 2.e transforms the *localcall* non-terminal into a block, and then the loop can be parsed by rule 5 yielding *b2*. The attribute *demand* for *b2* is computed, by multiplying the demand of the loop body with the number of repetitions. In step 3 the *loop* is rewritten as a block (rule 2.c) and then rule 3 collapses the sequence of blocks (*b2, a4)* into *b3*.

This node, obtained by parsing a complex structure, will be mapped to a *single* Klaper element (also given in grey in Fig. 2.c). A simplified Klaper model of the component ECommServer offering the service Buy, is described by the graph of steps from *start* to *end* given in Fig. 2.c. After the service, the component will remain *idle*.

The example shows that the abstraction-raising transformation from UML to Klaper aggregates away details that are not important for performance/reliability analysis, but maintains enough information so that the analysis results (such as response times for services under different workloads, throughputs, utilization of different resources, queue lengths, time to failure, etc.) can be imported back in the UML models, by using the mapping between the elements of the source and target models. The example also illustrates how the graph grammar rules can be used to impose and verify additional well-formedness constraints on top of the standard UML metamodel.

4 Conclusions

This paper tackles the problem of abstraction-raising transformation for deriving analysis-oriented models from design specifications of component-based software systems. The proposed approach addresses the need to bridge the significant semantic gap that usually exists between the software design domain (source) and the performance/reliability domain (target). We propose to separate the concern of parsing the source model for extracting higher-level of abstraction concepts from the concern of mapping between the source and target model, which could be realized by traditional MDA techniques. A graph grammar is used to parse the source model and to extract higher-level of abstraction constructs that are semantically closer to the target domain. Our proposal can be seamlessly integrated into standard MOF-based transformation frameworks, as the parsing and the extension of the source model can be realized as a pre-processing step of a “conventional” model transformation pipeline.

Acknowledgements

This work was done during Antonino Sabetta's visit to Carleton University, Ottawa, with the financial support of NSERC Canada through its Discovery and Strategic Grants, of the MIUR-FIRB project "PERF: Performance evaluation of complex systems" and of the MIUR project "Model driven design and analysis of adaptable software architectures".

References

1. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M., "Model-based performance predic-tion in software development: a survey" IEEE Transactions on Software Engineering, Vol 30, N.5, pp.295-310, May 2004.
2. S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable Petri net models," in Proc. of 3rd Int. Workshop on Software and Performance (WOSP02), Rome, July 2002, pp. 35-45.
3. C. Cavenet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens, "Analysing UML 2.0 activity diagrams in the software performance engineering process," in Proc. 4th Int. Workshop on Software and Performance (WOSP 2004), Redwood City, CA, Jan 2004, pp. 74-83.
4. V. Cortellessa, A.Pompei, "Towards a UML profile for QoS: a contribution in the reliability domain", In Proc. 4th Int. Workshop on Software and Performance WOSP'2004, pp.197 - 206, Redwood Shores, California, 2004
5. K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches", OOPSLA'03 Workshop on Generative Techniques in the Context of MDA, 2003.
6. V. Grassi, R. Mirandola, A.Sabetta, "From Design to Analysis Models: A Kernel Language for Performance and Reliability Analysis of Component-based Systems", In Proc. 5th Int. Workshop on Software and Performance WOSP'2005, pp. 25-36, Palma, Spain, July 2005.
7. J. Jürjens, P. Shabalin, "Automated Verification of UMLsec Models for Security Requirements", Proceedings of UML 2004, Lisbon, Portugal Oct. 11–15, 2004.
8. J.M. Kuster, S. Sendall, M. Wahler, "Comparing Two Model Transformation Approaches", Proc. Workshop on OCL and Model Driven Engineering, October, 2004.
9. T. Mens, K. Czarnecki, P. Van Gorp, "A Taxonomy of Model transformations", in Proc. of Dagstuhl 04101 Language Engineering for Model-Driven Software Development (J. Bezivin, R. Heckel eds), 2005.
10. D.C. Petriu, H.Shen, "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", in Computer Performance Evalua-tion: Modelling Techniques and Tools, (T. Fields, P. Harrison, J. Bradley, U. Harder, Eds.) LNCS 2324, pp.159-177, Springer, 2002.
11. D. C. Petriu, C. M. Woodside, "Performance Analysis with UML," in UML for Real, (B. Selic, L. Lavagno, and G. Martin, eds.), pp. 221-240, Kluwer, 2003.
12. OMG, QVT-Merge Group, "Revised submission for MOF 2.0 Query/Views/Transformations RFP", version 1.0, April 2004.
13. OMG, "MDA Guide", version 1.0.1, June 2003.
14. OMG, "UML Profile for Schedulability, Performance, and Time", version 1.0, formal/03-09-01, September 2003.
15. OMG, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QoS)", Adopted Specification, ptc/2004-06-01, June 2004.
16. Schürr, A., Programmed Graph Replacement Systems, in G.Rozenberg (ed): Handbook of Graph Grammars and Computing by Graph Transformations, pp. 479-546, 1997.
17. Woodside, C.M, Petriu, D.C., Petriu, D.B., Shen, H, Israr, T., and Merseguer, J. "Performance by Unified Model Analysis (PUMA)", In Proc. 5th Int. Workshop on Software and Performance WOSP'2005, pp.1-12, Palma, Spain, July 2005.