A Pattern-Based Approach to Model Software Performance*

José Merseguer
Dpto. de Informática e Ingeniería de Sistemas, University of Zaragoza, Spain
jmerse@posta.unizar.es

Javier Campos
Dpto. de Informática e Ingeniería de Sistemas, University of Zaragoza, Spain
jcampos@posta.unizar.es

Eduardo Mena
Dpto. de Informática e Ingeniería de Sistemas, University of Zaragoza, Spain
emena@posta.unizar.es

ABSTRACT

The use of the object-oriented paradigm in the software industry is nowadays a reality. Approximations like frameworks, components, workflows or patterns are gaining place, sometimes to complement object-oriented development. All these approaches, as well as the object-oriented paradigm, make special emphasis on the reuse of the software as a way to increase productivity. But it must be admitted that sometimes the performance of the deployed software systems is not as good as desired. Thus, techniques and methods to predict performance are subject of research, and Software Performance Engineering is concerned about them. In this article, we present an approach based on patterns to develop performance models for software systems in the early stages of the software development process. Moreover, the approach has as a goal the use of formal models to predict performance. This work complements the proposal suggested in [10].

Keywords

Software performance, Petri nets, patterns, UML, software reuse, object-oriented paradigm

1. INTRODUCTION

It is widely recognized that software design is a hard task that requires a significant amount of effort and experience. In the last decades, several paradigms have been proposed in order to facilitate the *software development process*, for instance, pragmatic object-oriented methodologies such as OMT [13], OOSE [6]. Some of them have recognized and identified as goal the necessity of *reuse* in all the stages of the development process, specially in the analysis and design stages. So, software reuse [7] has become a key to increase productivity. Concerning software reuse, object-oriented approach represents the main paradigm to develop software, because the concepts underlying object-oriented paradigm promote reuse in all stages of the software life cycle. In the last years, reuse at the design stage in the object-oriented paradigm has been subject of study, obtaining interesting proposals such as the well-known *design patterns* [4]. Thus, design patterns appear as an interesting approach to assist software engineers to model software systems.

Moreover, the necessity to predict the *performance* of software systems is doubtless. Even nowadays, systems are frecuently deployed without the performance expected by clients. It is a common practice of developers to test the performance of their systems only when they have been implemented. Software performance field [16] is concerned about these problems. This discipline proposes the study of the system performance inside the software development process, but paying special attention in the early stages, when the proper actions to solve performance problems could we performed with less effort and less economical impact. Several approaches in this area have been proposed with success [15, 17, 9].

Our objective is to bring together both disciplines, software reuse and software performance. The object-oriented paradigm will be the key to combine both. Thus, we propose a process to assist the software requirements engineer in the construction of the system models. In the proposed process, the software engineer will be able to capture functional requirements as well as performance requirements. In order to accomplish reuse, the process promotes the use of patterns. This process, called PROP[1], is also concerned with the use of the adequate techniques to obtain performance results from the developed models.

The language proposed in [4] to describe design patterns deals with the structural and behavioural aspects of the software and also gives trends to accomplish the implementation phase. PROP proposes the enrichment of this language with performance features, describing for each pattern its performance goals and workload definitions, obtaining "patterns with performance"[2]. In [4], the OMT [13] notation is used to describe the structure and behaviour of patterns. We advocate for the use of the UML notation [2] because of its

*This work has been developed within the project TAP98-0679 of the Spanish CICYT.

[1]Performance Requirements for Object-oriented design using Patterns.

[2]We mean by "patterns with performance", design patterns augmented with time annotation to evaluate performance.

widely acceptance between the software engineering community. UML will be enriched with performance annotations, we will use the proposal stated in [10]. Another proposals to annotate time aspects in UML have been stated for the real-time field [3], therefore with different objectives.

Using patterns, with the corresponding performance annotations, in conjunction with the guides given by PROP, the engineer will be assisted in the construction of the complete performance annotated UML diagrams for the system. In order to obtain numerical results from the UML annotated diagrams for the complete system, we promote the use of formal analytic techniques such as those surveyed in [14]. However UML lacks of the necessary formalism to accomplish it, due to the pragmatic view point used to develop UML diagrams. Therefore, the models of the system must be translated into a formal model, such as *Petri Nets* [11]. After that, we have the possibility to perform the desired performance analysis. We propose to maintain UML notation as a friendly front-end for the designer, so the translation into the formal model must be transparent for her/him. In this way, our approach combines *pragmatism and formality*.

Notice that this approach offers the possibility of reusing performance patterns, in the same way as traditional design patterns offer the possibility to reuse design and code. The approach could be applied to any kind of software design. We are specially interested in distributed systems design. In these environments, performance could be specially important because of the possible improper use of net resources. We specifically study the use of the *mobile agent technology* [12] to design and implement the systems. Thus, we are primarily interested in the study of patterns that fit well for this kind of software.

The rest of the paper is organized as follows. Section 2 introduces performance features in design patterns in order to achieve PROP patterns. Section 3 shows the process proposed by PROP to model performance and functional requirements, in a pragmatic way, including the translation from the pragmatic models into the corresponding formal ones, and the techniques used to solve the formal model in order to obtain performance results which aid the designer to take decisions. Finally, concluding remarks are presented in section 4.

2. PROP PATTERNS

Design patterns are defined in [4] as *"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"*. Benefits from design patterns, as it was suggested, come from their ability to achieve software reuse. In the same way, we claim for reuse in performance modelling. In section 2.2 we explain how patterns and performance could be a good marriage.

2.1 Basics on design patterns

We assume that the reader is familiar with patterns language as proposed in [4], but we consider interesting to succinctly remember it. In this language, a design pattern is described using the next *sections*:

- **Pattern name**: the essence of the pattern succinctly.
- **Intent**: the problem that the pattern address.
- **Motivation**: a scenario that illustrates how the pattern solves the problem.
- **Applicability**: situations in which the pattern can be applied.
- **Structure**: representation of the classes in the pattern (using OMT notation).
- **Participants**: classes/objects participating in the design pattern and their responsibilities.
- **Collaborations**: how the participants collaborate.
- **Consequences**: trade-offs of using the pattern.
- **Implementation**: techniques to be used when implementing the pattern.
- **Sample code**: code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.
- **Known uses**: examples of the pattern found in real systems.
- **Related patterns**: patterns closely related to this one.

Our approach does not pretend to be orthogonal to design patterns. On the contrary, it pretends to integrate both perspectives. So, we propose to extend the pattern language from two sides: 1) some of the language current sections will be enhanced to support performance requirements; and 2) new sections will be added to the language with the same purpose. These improvements will be explained in the following.

2.2 Adding performance features to design patterns

In this section, we propose the enhacement of the design patterns language in order to describe performance features, so we obtain "patterns with performance". First, we propose the improvement of the "Collaborations" and "Participants" sections. Later, new sections for the design patterns language are proposed.

The "Collaborations" section of the language is enhanced as follows. Currently, this section is described in some patterns using a sequence diagram, the rest of the patterns describe it in a textual way. We propose the use of a sequence diagram in all the patterns to describe the section. In this way, a sequence diagram will be used to annotate the message load among objects. Also, the probability for the guards success will be annotated.

The meaning of the annotations in the diagrams, the techniques to obtain them and other details encountered to develop the process are largely explained in [10], so we do not extend here on them. As an example, the reader may notice in Figure 1, taken from [10], the annotations in bold face. Some of them represent the load of the parameters in the messages sent among objects (for instance 100K associated

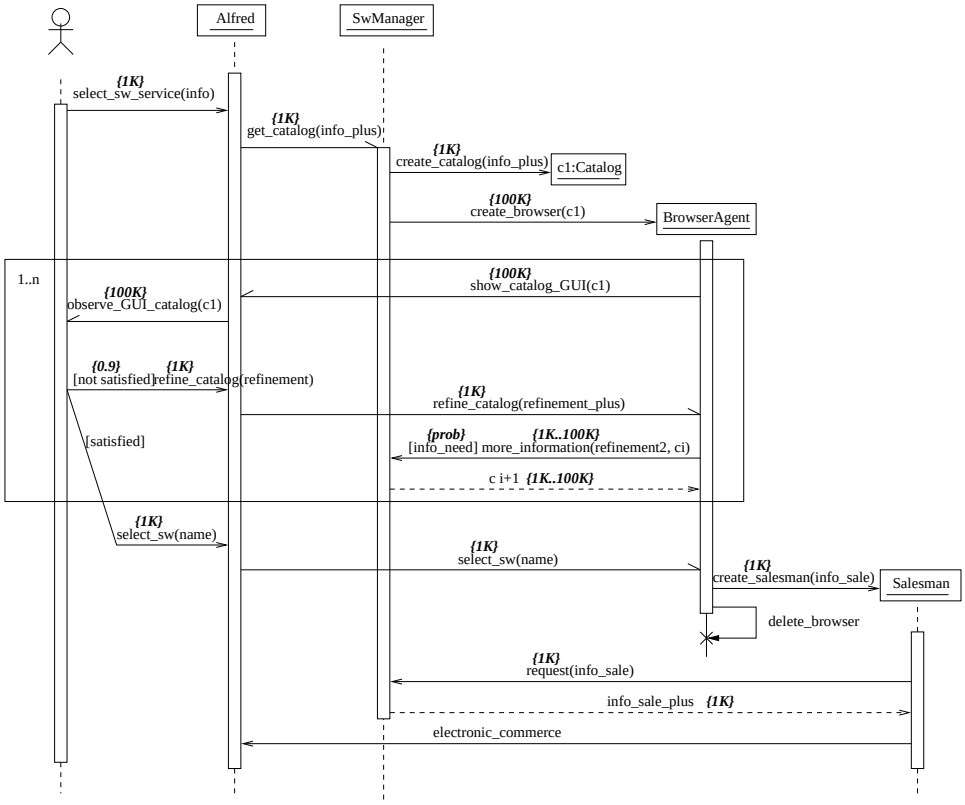

Figure 1: Annotated sequence diagram.

to create_browser message) and others represent the probability the messages success (such as 0.9 in the not_satisfied guard).

The "Participants" section, which describes the classes/objects participating in the pattern, must be also enhanced. A state transition diagram (STD) for each participant must be modelled. These diagrams represent the life of the objects. The STD will be annotated with the events load, the probabilities of the guards and the time to perform the actions. Figure 2, taken also from [10], shows an example that describes the way in which annotations are done. As it occurs with the sequence diagram, the complete description of the process to obtain the annotated STD and the meaning of the annotations are given in [10].

Now, we explain the proposal to extend the design patterns language with new sections. These sections taken from [16] are necessary to obtain a complete description of the pattern performance features:

- **Performance goals**: the pattern performance objectives. For instance, response time, throughput or utilization.
- **Workload definitions**: such as request arrival rates or the number of concurrent users.

The sections proposed in [4] together with the enhancement of the "Colaboration" and "Participants" sections and the new sections proposed, define what we call *PROP pattern*, to distinguish it from the design patterns.

The catalog presented in [4] contains twenty three design patterns. It would be very interesting to introduce performance in all of them. So, they could become PROP patterns. At the moment, we are only concerned about those which are appropriate for a concrete software technology: mobile agents. These are: *mediator*, *observer*, *proxy* and *facade*. We also consider the possibility of creating new patterns with their own performance features, but this will be the subject of future works.

The way to use a PROP pattern is the same as a design pattern. When the software engineer detects that the use of a pattern is possible in the system s/he is modelling, s/he develops, using the techniques given in [10], the annotated sequence diagram and the annotated state transition diagrams for the collaborating objects. At the moment, only a "piece" of the system has been modelled. The "piece" is ready to be used in the PROP process. Section 3.1 explains how to use these pieces in the PROP process.

3. THE PROP PROCESS

In this section, we present the PROP process; from now on and for simplicity we refer to it only as "the process". The main goal of the process is to assist software engineers to obtain the performance indices of the system that they are modelling. The process has also as objective to make use of well-known techniques from different fields: object-

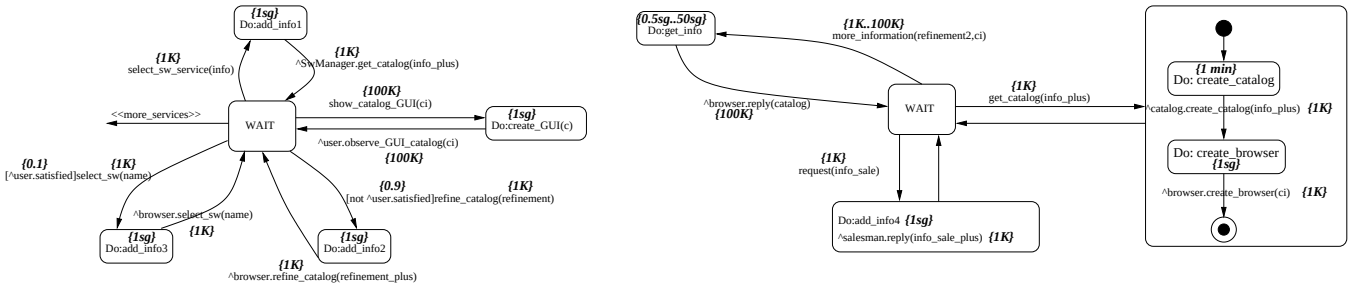

Figure 2: Examples of annotated STDs.

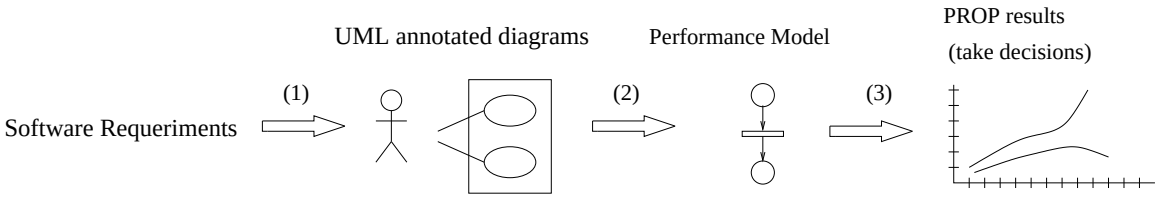

Figure 3: The PROP process.

oriented paradigm, performance modelling and performance evaluation.

Before illustrating the process in detail, we want to give a brief explanation of it. The process proposes to model the system in a pragmatic way using well-know object-oriented techniques: patterns as explained in section 2 will play a prominent role. As notation, we propose UML annotated with time; here our contribution in [10] can be used. When the pragmatic model is achieved, a formal model will be obtained from it. In the last step, the formal model will be used to obtain the performance indices using the proper techniques.

The following three main steps show the process, Figure 3 refers to them:

1. Model software requirements using annotated UML in conjunction with PROP patterns.

2. Use PROP translation rules to generate the corresponding formal model.

3. Use the techniques proposed by PROP to solve the formal model.

In the following sections, the process is described in detail.

3.1 Modelling system requirements

The objective of this section is to show the process to fulfil the UML annotated diagrams, from the software requirements, when a system is being modelled.

UML designers should model the system from two complementary perspectives, static view and dynamic view. System statics deals with system information structure, UML provides the class diagram to accomplish it. To express system dynamics and UML gives the possibility to use five different and complementary kind of diagrams: use cases, sequence diagrams, collaboration diagrams, state transition diagrams and activities diagrams. At this stage UML implementation diagrams are avoided.

In the object-oriented design process, the engineer must study the requirements of the system in order to identify where to use PROP patterns. Whenever the use of a pattern is possible, the engineer will complete the proposed sections. It is possible that s/he models only a few pieces of the system using patterns, or even none, depending on his/her experience and the system characteristics. Those requirements that have not been expressed using patterns must be designed inside the object-oriented design process. The engineer must develop the class diagram, the sequence diagrams and the STDs as it was proposed in [10].

3.2 PROP translation rules

At this point, we have modelled the system with UML notation, taking into account the load in the sequence diagram and the state transition diagrams. So, a pragmatic approach of the system has been obtained.

But this representation is not precise enough to express our needs (remember that we want to predict system performance). To accomplish it, we need to apply performance analytic techniques to the developed UML diagrams. But there is a lack in this field because no performance model exists for UML. To solve this lack, we have chosen stochastic Petri nets [1] as formal model, the reasons of the choice will be explained in the next section. It is our intention that the formal model can be obtained from the pragmatic model in a transparent way for the engineer. In [10] we gave the process and rules to accomplish the complete translation. In the following, we recall some of them succinctly.

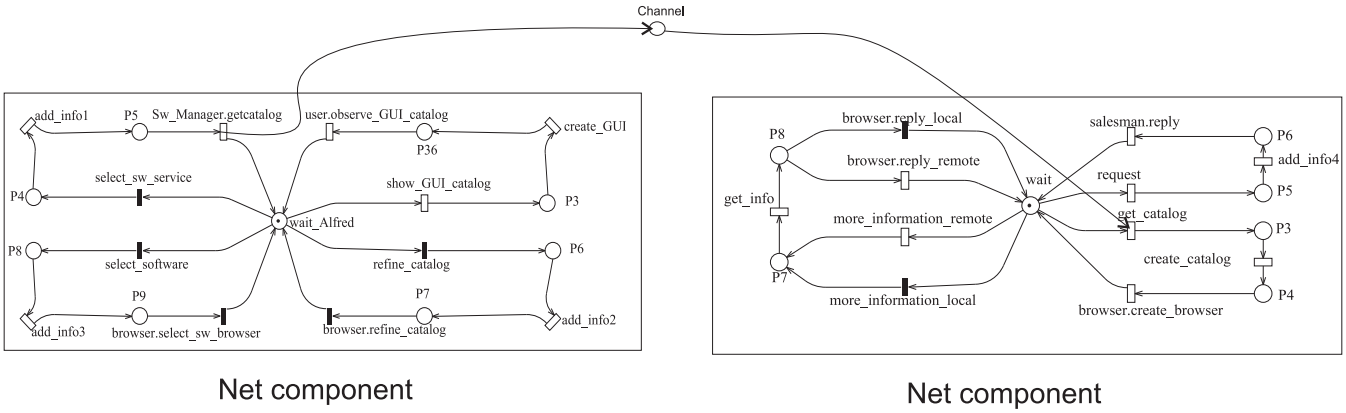

Figure 4: Asynchronous comm. between the two Petri net components obtained from the STDs in Fig. 2.

First, we must obtain a stochastic Petri net for each system class, the *component nets*. Obviously, the annotated STDs will guide us in this task. The transitions, actions, guards and states are considered to be translated into the Petri net. In Figure 4 component Petri nets for the STDs in Figure 2 are shown. They have been obtained using the rules proposed in [10].

Second, the sequence diagrams will be the guide to obtain a *complete* Petri net for the system using the previous component nets. For each message, there exist two transitions with the same name in two different component nets, the net representing the sender and the net representing the receiver (see for instance transition get_catalog in Figure 4). The way that we propose for the synchronization between transitions differs depending on the semantics of the message. Taking into account that UML distinguishes between messages with *wait semantics* and *no-wait semantics*, we propose a synchronous composition (by fussing transition instances) for wait semantics messages and asynchronous communication (through a channel place) for no-wait semantics messages. As an example, Figure 4 shows how the get_catalog transition, with no-wait semantics, synchronizes the components Petri nets. Proceeding in the same way the complete net for the system will be obtained.

The techniques proposed in the next section will be applied to the complete Petri net in order to obtain the system performance indices.

3.3 PROP techniques to solve the formal model

Now we describe the kind of analysis techniques can be applied to obtain performance goals for the system.

Several performance-oriented formalisms can be considered as an underlying mathematical tool for computing the performance indices of interest. Among them, queuing networks paradigm (QNs) [8], stochastic Petri nets (SPNs) [1], and stochastic process algebras (SPAs) [5]. For all of them, translation algorithms can be devised from the performance annotated UML diagrams in a similar way to those proposed in [10] for the case of SPNs.

SPAs, even if they are compositional by nature thus specially adequate for the modelling of modular systems, the present lack of efficient analysis algorithms and software tools for performance evaluation would make them difficult to use in real applications. We would suggest SPNs rather than QNs models because they include an explicit primitive for the modelling of synchronization mechanism (a *synchronizing transition*) therefore they are specially adequate for the modelling of distributed software design. Even more, a vast amount of literature exists concerning the use of Petri nets for both the validation of logical properties of the system (e.g., liveness or boundedness) and the quantitative analysis (performance evaluation).

Since we are specially interested in making use of modular design exploiting object-oriented approach, we refer to *net-driven decomposition analysis techniques* for SPNs as those surveyed in [14]. In that paper, a taxonomy for net-driven decomposition techniques is proposed, providing a framework for the consideration of a significant number of performance evaluation methods. In our present context, object life leads the definition of the "components" (subnetworks) of the Petri net, and this modular view of the model is crucial for the use of efficient net-driven based analysis techniques. In particular, exact solution, approximated solution, or bounds can be computed for the steady-state performance measures of the model using *divide and conquer* algorithms that take profit of the modular view of Petri nets.

Performance bounds are useful in the preliminary phases of the design of a system, in which many parameters are not known accurately. Several alternatives for those parameters should be quickly evaluated, and rejected those that do not satisfied response time requirements. Bounds become useful in these instances since they usually require much less computation effort. With additional computation investment, there is a large number of approximation techniques for stochastic Petri nets, and many of them make use of modular views of the model. Approximate values for the performance parameters are in general more efficiently derived than the exact ones. In this context, a pragmatic compromise to be handled by the designer of a system concerns the definition of faithful models, that may be very complex to exactly

analyse (what may lead to the use of approximation or just bounding techniques), or simplified models, for which exact analysis can be, eventually, accomplished. The general cost/accuracy trade-off must be taken into account in the selection of the desired approach.

4. CONCLUSIONS

The contribution of this work is the proposal of a process that integrates performance evaluation aspects in the object-oriented design of software applications using patterns. This paper complements the approach given in [10]. A performance extension of UML is used as a design notation. Design patterns are enriched with performance features, obtaining "patterns with performance". The resulting process, called PROP, is described in its main steps: Modelling business and performance requirements using annotated UML and patterns; translating UML diagrams into a performance model (stochastic Petri nets); and using specific analysis techniques (that make use of the modular and structured view of Petri nets) to get performance measures.

An important amount of work must be invested in the future:

- Performance aspects must be introduced in the existing catalogue of design patterns.
- Specific new patterns for particular applications could be proposed.
- The definition of a formal semantics for a subset of UML diagrams will permit an automatic translation into stochastic Petri nets (or any other performance oriented formalism).
- The existing analysis techniques for stochastic Petri nets that make use of decomposition of the model should be integrated in a tool and they would take profit of the object-oriented software design.

5. REFERENCES

[1] M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance Models of Multiprocessor Systems*. MIT Press, Cambridge, Massachussetts, 1986.

[2] G. Booch, I. Jacobson, and J. Rumbaugh. OMG Unified Modeling Language specification, June 1999. version 1.3.

[3] B. Douglass. *Real-Time Uml: Developing Efficient Objects for Embedded Systems*. Object Technology Series. Addison-Wesley, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic process algebras as a tool for performance and dependability modelling. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pages 102–113. IEEE CS-Press, April 1995.

[6] I. Jacobson, M. Christenson, P. Jhonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[7] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. Addison-Wesley, 1997.

[8] K. Kant. *Introduction to Computer System Performance Evaluation*. Mc Graw-Hill, 1992.

[9] P. King and R. Pooley. Using UML to derive stochastic Petri nets models. In J. Bradley and N. Davies, editors, *Proceedings of the Fifteenth Annual UK Performance Engineering Workshop*, pages 45–56. Department of Computer Science, University of Bristol, July 1999.

[10] J. Merseguer, J. Campos, and E. Mena. Performance evaluation for the design of agent-based systems: A Petri net approach. In *Software Engineering and Petri Nets (SEPN'2000) workshop within the 21st International Conference on Application and Theory of Petri Nets*, Aarhus, Denmark, June 2000. To appear.

[11] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[12] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1998.

[13] J. Rumbaugh, M. Blaha, W. Premerlani, E. Frederick, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, 1991.

[14] M. Silva and J. Campos. Performance evaluation of DEDS with conflicts and synchronizations: Net-driven decomposition techniques. In *Proceedings of the 4th International Workshop on Discrete Event Systems*, pages 398–413, Cagliari, Italy, August 1998. IEE Control.

[15] C. Smith and L. G. Williams. Performance engineering evaluation of object-oriented systems with SPE•ED^{TM}. In *Proceeding of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 135–154, St. Malo, France, June 1997. R. Marie, et al. eds. Lecture Notes in Computer Science, Vol. 1245, Springer-Verlag.

[16] C. U. Smith. *Performance Engineering of Software Systems*. The Sei Series in Software Engineering. Addisson–Wesley, 1990.

[17] M. Woodside, C. Hrischuck, B. Selic, and S. Bayarov. A wide band approach to integrating performance prediction into a software design environment. In *Proceedings of the 1st International Workshop on Software Performance (WOSP'98)*, 1998.