Department of Computer Science
University of A Coruña

PhD Thesis

From software architecture to formal verification of a distributed system

Author:

Juan José Sánchez Penas

Supervisors:

Thomas Arts (IT University of Göteborg)

Víctor M. Gulías (Universidade da Coruña)

July 2006

To Dores

Abstract

This thesis studies how to go from the software architecture to the formal verification of a distributed system.

As the motivation and target of our research, we use the *VoDKA* system, a distributed *VoD* server developed by the *LFCIA-MADS* research group using the *Erlang/OTP* platform. The software architecture of *VoDKA* is very flexible and complex, and better tools are needed in order to increase the confidence of the system architects and improve the overall system quality. We study how to use formal verification for that purpose.

Therefore, using several tools from the area of formal methods, we propose an innovative method for automatically extracting performance information about the system. As input to our method, we receive the system source code and the system configuration (the description of the components and how they interact). As output, we provide feedback information about the system performance and architectural bottlenecks. We extensively applied the method for analyzing the *VoDKA* system as a case study and showed how it can be reused with other tools and for other similar distributed systems.

Resumo (Thesis summary in Galician)

A presente tese propón un método e unhas ferramentas para ir dende a arquitectura software á verificación formal dun sistema distribuído.

O sistema distribuído que se estuda é o servidor de video baixo demanda *VoDKA*, desenvolvido no laboratorio *LFCIA-MADS* da Universidade da Coruña durante os últimos anos. *VoDKA* é un sistema cunha arquitectura complexa e moi flexible e escalable, que fai uso extensivo dos patróns de deseño e os componentes reutilizables. A tecnoloxía utilizada para o desenvolvemento é *Erlang/OTP*, unha linguaxe functional distribuida orixinalmente creada por Ericsson para a programación de sistemas de control.

Para a análise deste sistema, a tese desenvolve un método e utiliza unha serie de ferramentas mediante a aplicación da verificación formal. O obxectivo final deste método é a extracción de información útil sobre a arquitectura do sistema, de forma automática, tomando como punto de partida o código fonte e a descripción da configuración do sistema.

A tese está dividida en tres partes principais. Na primeira, motívase a investigación levada a cabo: *VoDKA* é un sistema innovador, que comparte algunhas características con sistemas semellantes distribuídos, e representa un reto moi interesante para a aplicación de métodos formais. Xusto despois indícanse os principais obxectivos da tese: coñecer máis e mellor o software a estudar, propoñer ferramentas para a mellora da súa arquitectura e estudar as posibilidades e límites do uso de métodos formais para a verificación de propiedades non funcionais dun sistema distribuído. Finalmente son introducidos os principais conceptos necesarios para a comprensión detallada do resto da memoria da tese: explícanse en detalle as ideas relacionadas co desenvolvemento de sistemas distribuídos utilizando patróns de deseño, e introdúcense os conceitos relacionadas co model checking e as álxebras de procesos.

Na segunda parte da tese, o servidor *VoDKA* é estudado en detalle co obxectivo de amosar como varias das características innovadoras do sistema convidan á utilización de métodos formais para incrementar o coñecemento do mesmo que teñen os membros do equipo de desenvolvemento. En primeiro lugar, descrébense os requisitos máis importantes dos servidores de video baixo demanda, e ao mesmo tempo fálase do estado da arte na área e das alternativas que inspiraran a *VoDKA* no momento da súa concepción. *VoDKA* é unha opción comparable ás alternativas comerciais e académicas, e especialmente útil en contextos onde a flexibilidade, escalabilidade e baixo coste son requisitos prioritarios. Despois disto, a arquitectura do sistema descrébese en detalle usando a aproximación do 4+1 Model, ofrecendo deste xeito distintos puntos de vista complementarios do sistema. Os patróns

de deseño, os componentes do sistema, e a API homoxenea que todos eles comparten son descritas en detalle nestes capítulos. A configuración de *VoDKA* pode variar enormemente dependendo das necesidades de cada despregue, co que na tese descrébense varios exemplos en redes de distinta natureza. O capítulo 6, resume as conclusións desta parte da tese: *VoDKA* funciona moi ben e ten unha arquitectura moi flexible, pero debido a isto e ás limitacións de *Erlang* (que é unha das chaves do éxito do sistema pero ten algúnas desvantaxas), manifféstase unha necesidade de mellores ferramentas que permitan aos deseñadores e arquitectos do sistema tomar as decisións arquitecturais correctas.

Finalmente, na terceira parte da tese, partindo do coñecemento adquirido sobre a arquitectura de *VoDKA*, descrébese como os métodos formais poden ser unha resposta á devandita carencia de ferramentas para a extracción de información sobre sistemas distribuídos. No capítulo 7 explícanse os primeiros resultados: por que e como se decidiu utilizar métodos formais para o proxecto *VoDKA*. A principal razón e a búsqueda dun incremento na calidade do sistema ofrecendo á equipa de desenvolvemento unha mellora na información sobre o rendemento do sistema que se pode extraer de xeito automático. De entre as posibles técnicas, óptase por utilizar *model checking*, ainda que a simulación, as probas automáticas e mesmo a proba de teoremas teñen vantaxas e problemas que indican que poderían ser utilizadas de xeito complementario. A proposta é a *verificacion formal áxil* realizada en cada iteración do ciclo de vida do proceso de desenvolvemento de software co obxectivo de enriquecer as decisións tomadas sobre a evolución da arquitectura do sistema.

A aproximación áxil baseada en *model checking* que se propón nesta tese é descrita en detalle no capítulo 9, e consiste na definición dun método que recibe como entrada o código fonte do sistema e a descripción da configuración do mesmo, e é quen de extraer de xeito automático información sobre o rendemento utilizando fórmulas lóxicas que son verificadas contra o grafo de estados do sistema.

Para a implementación do método, son necesarias ferramentas concretas. No capítulo 8 descrébense as ferramentas escollidas en detalle. Algunhas delas son ferramentas avanzadas que xa estaban disponibles, e outras foron adaptadas ou desenvolvidas *ad hoc* para as necesidades do método. Na primeira versión do método proposto, primeiro faise unha transformación do código fonte *Erlang* a unha especificación no álxebra de procesos μ`CRL`; para isto, o compilador `etomcrl` foi desenvolvido. O compilador utiliza os patróns de implementación de *Erlang*, chamados *behaviours*, como base para simplificar a traducción abstraendo os detalles de baixo nivel. `etomcrl` é quen de traducir automáticamente gran parte da linguaxe *Erlang*. Para a simulación dos usuarios que fan as peticións ao servidor de video, utilízase o non-determinismo do álxebra de procesos. Unha vez temos a especificación, pódense utilizar as ferramentas do μ`CRL toolset` para a xeración do grafo de estados do sistema. Para poder manexar grafos grandes, faise unha reducción antes de realizar a verificación por *model checking*. Para a reducción e a verificación utilízase `CADP`.

O método proposto foi aplicado satisfactoriamente para a análise do servidor *VoDKA*. Seguindo a aproximación explicada, fomos quen de obter información como o número de veces que unha película pode ser visualizada antes de que o servidor se sature, ou o número de veces que se pode ver unha película A sendo ainda posible visualizar outra película B a posteriori; seguindo o mesmo método, tamén

podemos extraer información sobre a arquitectura do sistema, os bottlenecks, ou mesmo proporcionar aos deseñadores pistas sobre que decisións arquitecturais tomar cando un novo compoñente é introducido no sistema. Os resultados obtidos son moi interesantes e a maior parte dos obxectivos plantexados ao comezo da investigación foron alcanzados. Porén, a aproximación ten limitacións que son descritas en detalle ao longo da tese: algunhas delas inerentes ao uso de técnicas baseadas en *model checking* (como é o caso dos grafos de estados infinitos ou moi grandes) e outras derivadas de que as propiedades sobre o rendemento do sistema non sempre son fácilmente formualizables como expresións lóxicas sobre o grafo de estados.

Ademáis das limitacións comentadas, as ferramentas escollidas tamén teñen algunhas limitacións, como o tempo necesario para a xeración do grafo completo, ou a dificuldade para obter a reducción axeitada. Co obxectivo de superar estas limitacións, e ao mesmo tempo de amosar que o método proposto non está ligado a unhas ferramentas concretas, repetiuse outra vez o estudo sobre *VoDKA* utilizando agora `McErlang`, que permite a realización de *model checking* directamente sobre o código fonte *Erlang*. Explícase como `McErlang` pode ser utilizada tanto para a xeración do grafo de estados completo, similar ao que se fai coas μ`CRL toolset`, como para verificar directamente as propiedades tal e como se facía con `CADP`. `McErlang` non é tan madura como as ferramentas utilizadas anteriormente, pero o feito de traballar directamente co código e a máquina virtual *Erlang* dalle moita potencia á hora de acceder a calquera tipo de dado que está manexando o sistema en calquera dos estados intermedios do grafo que se está a construir. Isto proporciona unha potencia maior que o razoamento sobre accións e estados sen información, que se realizaba na aproximación anterior. Usando `McErlang`, fomos quen de mellorar os resultados obtidos coas ferramentas anteriores e de abrir novas posibilidades para a investigación futura.

Outro dos obxectivos da tese era a creación dun método que poda ser utilizado sen necesidade de ser un experto na área dos métodos formais. Para isto, por riba das ferramentas especializadas utilizadas, construimos o prototipo dunha aplicación de alto nivel que oculte todos os detalles formais que non sexan necesarios durante o proceso.

Tamén é interesante resaltar que o método proposto non é só flexible nas ferramentas utilizadas -como xa se amosou- senón que tamén pode ser aplicado a outros sistemas distribuídos. Nesta tese o obxectivo central era probalo realizando a extracción de propiedades do servidor *VoDKA*, pero como se explica no capítulo 9, pode ser aplicado a outros sistemas similares, que compartan as principais características de *VoDKA*, principalmente: ter unha arquitectura flexible, ter consciencia do hardware sobre o que se executa a aplicación, e usar frecuentemente patróns de deseño como base para a creación de componentes software.

Os resultados da tese son satisfactorios, e numerosos camiños para o traballo futuro, tanto no ámbito do servidor *VoDKA* como na mellora das ferramentas de extracción de información quedan abertos.

Acknowledgements

I have passed different periods while doing this thesis, and I really need to thank all the people that helped me to overcome the bad moments and to enjoy and appreciate the good ones.

First, I would like to thank Thomas Arts. It can sound like a commonplace, but I truly think that without him, this dissertation would not have been possible. I still remember the first time I visited him at the Ericsson's Computer Science Laboratory, in Stockholm, the summer of 2001, looking forward to starting some collaboration with him in the fields of distributed programming and software verification. Lots of things have happened since then, and I have never stopped learning from his insights and suggestions. I do not have words to thank his enthusiasm and support during all these years. Thanks also for the afternoons with Torkel, Dagmar and Heleen, with the snow outside.

I would like to thank Víctor Gulías for supervising the thesis and giving me support and freedom (two things difficult to combine) during the latest years. I also remember how excited I was when a lot of years ago, after one of the last exams of the degree, he suggested me to become part of the *LFCIA-MADS* research group. I have learned a lot from the experience and I am very happy to have accepted his invitation.

I sincerely thank José Luis Freire for his knowledge and humanity, and all the people I worked with at the *LFCIA-MADS* group and the *VoDKA* development team.

During my several visits to Sweden I had the opportunity to meet very nice people. In 2001, when I first visited Thomas, he was working with Clara Benac Earle; since then, I have had the pleasure to collaborate with Clara in several subjects related to this thesis. It was also a pleasure to meet Lars-Åke Fredlund, from who I learned a lot, and not only about theorem proving and `McErlang`. I look forward to doing more research with them. In general, I feel lucky for being able to met new people at the Ericsson's CSLab (thanks to Bjarne Däcker for redefining the concept of boss), the Swedish Institute of Computer Science (thanks to Mads Dam for kindly hosting me there), and the IT-University of Göteborg.

Thanks to the *Erlang* community, and the authors of the μ`CRL toolset` and `CADP`, for being so kind answering my doubts and concerns.

Thanks also to my fellows at the Igalia project. Five years ago I started with them a big adventure, which has been one of the most interesting professional experiences in my life.

I have to thank my parents for their love and unconditional support, and for teaching me to fish instead of (only) feeding me fish. Also to my brother Rubén

(for that e-mail pending to be answered), and to Sonia (and the no-name baby) and the lovely Ledicia, who still does not have a clue on what a Ph.D. thesis. Also need to thank Jose, Loli and Xosé, for making so easy to become part of their family, as if it was the most natural thing in life. I have to mention also my grandparents, specially Emilio, who passed away one year ago and comes to my mind very often.

Thanks to Alex for being a Friend, always there. I am more than happy for that decision about moving to the front of the classroom in the second year at the University.

Thanks to the rest of my friends, María and Manolo (and now Mateo, for the relax-moment in our sofa), Ana (for those hugs), Vanesa and Toni (for the escapes to Barcelona), Ángel (for the energy in Alfajore pills), and the rest of the people who share the hibiscus-experience in `canceladeafora10`.

And finally, very special thanks to Dores, first for her love and patience, because this thesis is more hers than mine (the cover should truly have her name written), and second and more important, because (and I am not a friend of saying this things in public) **nothing** in life would make sense without her.

Contents

List of Figures

Part I

Context

The first part of the thesis explains the context needed for understanding the research that has been carried out. In the three following chapters we first motivate the work, then we describe the main goals and contributions of the thesis, and finally we describe all the preliminary knowledge needed in order to understand the rest of this manuscript.

Chapter 1

Motivation

During the last ten years, the *LFCIA-MADS* (Models and Applications of Distributed Systems) research group has been working in innovative ways of developing distributed systems. The *VoDKA VoD* server[1] is an example of such a system, first developed by the group as an innovative flexible component-based streaming architecture and since 2003 offered as a commercial product by a spin-off company[2] developed from the *University of A Coruña*.

VoDKA has been developed using *Erlang/OTP*, a platform first created by the telecommunications company Ericsson as an internal product and released as *Open Source* in 1998. *Erlang* has increased its popularity during the last decade and it is used nowadays for developing distributed applications all over the world. *Erlang*-like technologies have more and more importance in modern world, since fault-tolerance, concurrency, maintainability or time-to-market are requirements for almost all kind of systems.

Despite the innovative aspects of the *VoDKA* system, it shares some characteristics with a considerable part of the distributed systems that are developed nowadays, specially, of course, with those that use *Erlang/OTP* as underlying platform.

Since the first stages of the *VoDKA* project, the development team was confronted with several difficulties, raising different kinds of research questions. In these thesis we focus on one of those difficulties: how to automatically obtain information about the system performance from its complex software architecture.

If we understand *performance* in a general way, we can include things like the system capacity, the possible bottlenecks in the architecture, or inconsistencies between requirements and software configurations. Doing the study by traditional ways -mainly by deploying the system and testing it- turned out to be a time-consuming process, and the challenge of this thesis was to find alternatives to them.

The alternative we wanted to explore is formal methods, a promising approach that has been gaining importance dramatically during the last years in parallel with the development of new and more complex distributed systems. From being mostly a pure research area, only used successfully in the fields of hardware or protocol verification, it has grown into a richer field where all kind of new approaches, tools

[1] *VoDKA* is a Spanish trademark of LambdaStream Servicios Interactivos, S.L.
[2] LambdaStream Servicios Interactivos, S.L. (http://www.lambdastream.com)

and methodologies are appearing.

In 1996, in the ACM Computing Surveys [CW96], the situation was described by Clarke and Wing in the following way: *Finally, we believe that current techniques and tools for specifying and verifying systems have advanced to the point that with another "turn of the crank" partial analysis is tractable for real, industrial-sized, complex computer systems. This last claim suggests that the research community is potentially on the brink of a major breakthrough. Thus, by forging ahead we hope that we can provide the next generation of formal methods–their notations, techniques, and tool support–and that they will be used routinely by the software and hardware engineers of tomorrow.*

Agreeing with that definition, and considering it to be still valid nowadays, the main motivation of this thesis is to explore how one should go from the software architecture of distributed systems like *VoDKA*, to extract interesting system properties using formal verification. In other words, the thesis explores the use of techniques from the area of formal methods in order to increase the quality of a given kind of distributed systems, exemplified by the case study of *VoDKA*. The method we propose allows the engineers to predict the behaviour of the system and therefore to learn more about the software architecture in order to improve its performance.

The research carried out in this thesis builds therefore upon the tradition already present in 1996 and provides new methods and tools for advancing in that direction.

The work carried out includes studying the state of the art in the field; analyzing the system we are going to use for the case study, its development and its problems; and proposing a complex set of tools (some of them developed after finding a concrete need) and a method for analyzing systems like *VoDKA* and extracting performance properties. The method and tools are used for the case study (analyzing a distributed *VoD* system), but they are explained as generic solutions that can be reused in other kinds of systems that share with *VoDKA* the basic features.

Chapter 2

Introduction

Contents

As already introduced in the previous chapter, the main goal of this thesis is to carry out research about the use of formal methods in order to extract properties from the software architecture of distributed systems. In the current chapter, we first describe with more detail the goals of the thesis; then, we enumerate the main contributions and cite the main articles that were published with thesis material; and finally, we describe the structure of the rest of the manuscript.

2.1 Main goals of the thesis

The thesis goals are derived from what we have explained in the motivation, and most of them have derived in contributions to the current state of the art in the field. Hereafter we describe some of them:

- Contribute to the *VoDKA* development creating methods and tools for knowing more about the system. The tools should help in the process of extracting relevant information about the software architecture.
- Analyze in detail the *VoDKA* project itself and learn more about its architectural ideas and components, and about how the system evolved through the years. The proposed solutions should take the characteristics extracted from the study of *VoDKA* into account.
- Explore the possibilities of formal methods for improving the design of distributed systems giving feedback to the developers. This contributes to the idea of the formal development of a complex distributed system.
- Look into verifying non-functional aspects of distributed systems using automatic model checking techniques. Traditionally, model checking techniques

have been used for correctness properties, and the goal here is to extend that usage to other behavioural properties and specially to performance information.

- Explore the tools from the area of formal methods and their capacity for being used with real distributed systems. Test the most interesting ones and complement them with the required features, new usages, or new components.

- Extract the information automatically, reducing as much as possible the human interaction. In this sense, using more automatic approaches, like model checking, will be a priority.

- Extract as many properties as possible from the *VoDKA* system, always trying to take into account how general they are and how easy could be to apply them to similar distributed architectures.

2.2 Main contributions of the thesis

- Study of the technologies and the state of the art in the areas related to the content of the thesis. Developed in Chapter 3.

- Detailed description of the *VoD* servers and their main requirements, and study of the state of the art in *VoD* systems development. Developed in Chapter 4.

- Detailed description of the *VoDKA* architecture using the *4+1 Model*. Developed in Chapter 5.

- Study of the main problems found during *VoDKA* development and how they relate to the need of formal verification techniques. Developed in Chapter 6.

- Motivation and description of an approach for using formal methods in *VoDKA*, and answer to the main questions about the context of the approach and how it relates to software engineering. Developed in Chapter 7.

- Maintenance, improvement and creative usage of the `etomcrl` tool for translating *Erlang* programs to μ`CRL` specifications (shared contribution with Clara Benac Earle and Thomas Arts). The tool is described in detail in Chapter 8 and the usage in Chapter 9.

- Method for going **from software architecture to formal verification of a distributed system**. The method has several detailed phases and uses a set of well-known tools from the area of formal methods together with *ad hoc* tools. Besides, it focuses in the extraction of performance information from the source code, although it could be used for other kinds of properties. The method is described in the first part of Chapter 9.

- Detailed example of usage of the proposed method with *VoDKA* as a case study. Described in the second part of Chapter 9.

- Creative usage of the `McErlang` tool for offering an alternative to the tools in the method, extracting performance properties directly model checking the *Erlang* source code (shared contribution with Clara Benac Earle and Thomas Arts). As in the case of `etomcrl`, the tool is described in detail in Chapter 8 and its usage in Chapter 9.

- Development of a graphical user interface (*VoDKAV*) as a proof of concept for hiding all the low level formal details in the method proposed for extracting properties. The goal is to make usable the method for developers and designers that are not familiar at all with formal methods. The tool is described in Appendix C and its usage is shown during the method description in Chapter 9.

2.3 Published articles related to the thesis

In this section we will show the main publications related to the material presented in these thesis.

Articles related to the preliminaries and the state of the art:

- *State of the Art and Design of VoD Systems* (in Spanish). Published in the International Conference on Information Systems Analysis, SCI 2000 [SGVM00].

- *State of the art in formal verification of distributed functional software*. Published as a report for the Diploma de Estudios Avanzados, 2001 [Sól].

Articles related to the *VoDKA* project:

- *A Monitoring and Instrumentation Tool developed in erlang*. Published in the International Erlang User Conference, EUC 2000 [BGS00].

- *The Tertiary Level in a Functional Cluster-based Hierarchical VoD System*. Published in Eurocast 2001 and later selected for publishing in LNCS [BGSJ01].

- *Implementing a monitoring model for a Video-on-demand server in* Erlang (in Spanish). Published in the Conferencia Latinoamericana de Informática, CLEI 2001 [VGS+01a].

- *Using distributed functional programming and Linux clusters for the development of Video-on-demand servers* (in Spanish). Published in the Simposio en Informática y Telecomunicación [BGMS01].

- *Exploiting Sequential Libraries on a Cluster of Computers*. Published in the Erlang Workshop 2001 [BFGS01].

- *An* Erlang*-based Hierarchical Distributed VoD System*. Published in the International Erlang User Conference, EUC 2001 [SFB+00].

- *Functional Scheduling in a Distributed VoD Server*. Published in the International Workshop on the Implementation of Functional Languages, IFL 2003 [SBG+01].

- *Extending the* VoDKA *Architecture to Improve Resource Modeling.* Published in the Erlang Workshop 2003 [PR03].
- *Lambda goes to Hollywood.* Published in Practical Aspects of Declarative Languages, PADL 2003 [GAS03a].

Articles related to model checking and performance analysis:

- *Performance evaluation of a real-time multi-thread system using an asynchronous reactive model.* Published in the Simposio Español de Informática Distribuída [VGM+00].
- VoDKAV *Tool: Model Checking or Extracting Global Scheduler Properties from Local Restrictions".* Presented at the Third International Conference on Application of Concurrency to System Design, ACSD 2003 [SPA03].
- *Global Scheduler Properties from Local Restrictions.* Published in the ACM Sigplan Erlang Workshop 2003 [AS02].
- *Performance analysis using Model Checking.* Published in the International Erlang User Conference, EUC 2003 [PA03].
- *From* Erlang *to* `μCRL`. *Making industrial code available for research tools.* Published in proceedings of the Forth International Conference on Application of Concurrency to System Design, ACSD 2004 [TA04].

During the development of the thesis, the author also did some collaborations with other lines of research. Even though they are not directly part of the core of the thesis, they were interesting as source of inspiration or as a way of learning more about related topics. Examples of these publications are *Fusion and deforestation in Coq* [FNBFFBS01] or *Towards a Certified and Efficient Computing of Gröbner Bases* [SJGFS05], where the reasoning is done step by step using theorem provers. In other cases the collaborations are useful for getting more insight in one of the concepts most important in this thesis: the design patterns; in [GTBRS01], this concept was studied from the object oriented point of view. In order to avoid having only a very *Erlang* oriented point of view, it was interesting to explore ideas related to *Persistent Haskell* in [QS01]. Finally, other examples of using *Erlang/OTP* as development platform were explored during this years, helping on the way to make the method presented in this thesis more general. Examples of those projects are a distributed VLAN software switch [GCSPMR04], or a cluster-based payment gateway [AGFS02, AGF+03], both developed using the *Erlang* design and implementation philosophy.

In parallel with the finalization of the thesis manuscript, some new articles have been written, most of them heavily based on the contents of this document. One of the articles is based on the contents written for Chapter 8 and is being submitted for publication in an international journal in the area of formal methods and tools; other article is based on the contents of Chapter 9 and is also being submitted to another journal in the area of applied formal methods. Also, some of the contributions of this thesis have recently been or are going to be submitted to international conferences, as it is the case of the experiences on using `McErlang` in

the *VoDKA* system or the contents of Chapter 7 on the motivation and discussion of the use of formal methods for the *VoDKA* project.

2.4 Structure of the thesis manuscript

The thesis starts explaining the preliminaries needed in order to understand the rest of the manuscript. Concepts like *Erlang/OTP*, design patterns, the process algebras, or the state of the art in the area of formal methods for distributed software verification are presented in Chapter 3.

After that, the thesis manuscript is divided into two main parts.

The first one talks about the *VoDKA* project. In Chapter 4, the main concepts related to *VoD* systems, the common requirements, and the state of the art in their development are presented. After that, in Chapter 5, the *VoDKA* system is described in detail. Finally, in Chapter 6, the lessons learnt from the development of *VoDKA* are used in order to motivate the second part of the thesis.

The second main part talks about the verification framework that we propose for extracting properties from distributed systems. In Chapter 7, the approach is put into the context of the software engineering tradition. Chapter 8 presents all the tools that are needed in our method, some of them partially developed during this thesis. Finally, Chapter 9 explains the method proposed and all its stages, and applies it to *VoDKA* as a case study.

After the method is described in detail and the case study present, we discuss the main global conclusions about the thesis in Chapter IV.

Appendixes show some complementary information interesting in order to understand some details of the thesis. Appendix A shows a simple example of the translation from *Erlang* to μ`CRL` using the `etomcrl` tool and some extra details about the tool itself. Appendix B shows the original source code of the *VoDKA* model used as case study and the relevant parts of its translation to μ`CRL`. Finally, Appendix C shows the implementation details of the graphical user interface developed on top of the verification method proposed.

Chapter 3

Thesis preliminaries

Contents

This chapter is an introduction to the main concepts the reader needs to understand in order to understand the rest of the thesis. Apart from explaining some concepts and technologies, the state of the art in some of the fields is also presented.

3.1 *Erlang/OTP* and the design of distributed systems

The case studies and even some of the philosophical decisions in the work that is presented here are heavily influenced by the *Erlang/OTP* platform, an interesting example of a high level distributed functional language used for developing real

industrial systems. In this section, the main *Erlang* concepts, tools and projects are presented.

3.1.1 Short motivation and history of *Erlang*

During the eighties, inside the Computer Science Laboratory of Ericsson, a research project was started to select the best available programming language for telecommunication systems.

Telecom applications share very special requirements. They are normally very complex software embedded in powerful hardware. Besides, historically the downtime for classical telecommunication services like phone calls is almost zero, so the modern control systems for this kind of networks need to work 24x7. Maintainability is another key feature in a telecom-oriented system, because the software normally lives for a long time and the costs of maintaining it are the bigger ones at the end of the life cycle, so the ideal language or platform should support the creation of easy-to-maintain software. Also, concurrency and distribution are inherent features in systems that give support to services that a huge amount of users are accessing at the same time. Finally, fast prototyping and short time-to-market are also very important in a changing field where the ideas need to be quickly translated into products.

With all the above features in mind, a group of researchers at the Ericsson CSLab started to look into the languages and development platforms available at that time, checking if they satisfied all the needed requirements. The conclusion was negative: although some of them had interesting features, none gathered all of them. Therefore, the group decided to create a new language and the first experiments were carried out. The new language, named *Erlang* after the Danish mathematician Agner Krarup Erlang, who developed important studies about queuing theory at the beginning of the 20th century, that are widely used in the telecom world[1].

The initial versions of *Erlang* were developed by Joe Armstrong, Robert Virding and Mike Williams. The first abstract machine and the compiler where developed in Prolog. After several years improving the language, the first projects where started inside Ericsson. In 1995, what is still the main *Erlang* book [AVWW96] was published and, in the second part of the nineties the first big projects, some of them important Ericsson products, were developed using *Erlang*. In 1996, the first version of the Open Telecom Platform, a set of Erlang libraries, recommendations and design patterns, was released.

After using the language inside the company for more than one decade, in 1998 *Erlang* is released with an *Open Source* license. The important success of other *Open Source* projects like *GNU/Linux* and the need of a wider community in order to have a faster evolution in the language and better access to trained developers were possibly the reasons why the decision was taken.

Since then, the *Erlang* community has been constantly growing. Every year new projects and companies are started. The language is taught in universities, and has converted itself into a competitive option for developing not only telecom

[1]The alternative and possibly true story says that *Erlang* was just an abbreviation of ERicsson's LANGuage

systems but any kind of distributed control system.

More detailed information on the *Erlang* history can be found in [Dö0].

3.1.2 *Erlang* main features

Before describing the language features, it is interesting to remark the goals for which it was designed. *Erlang* is adapted to soft real-time distributed systems (there is not support for hard real-time where response times are exactly fixed). Also, the language is designed to support distributed problems, where the tasks can naturally be parallelized. In this aspect, it can be similar to other distributed languages like Occam, CSP or Concurrent Pascal.

The first *Erlang* syntax was similar to STRAND, but nowadays it looks more like a dialect of ML without types. The conceptual concurrency model that is used is very similar to the one in SDL (this is why still today some developers use diagrams that come from the programming environment of that language). One of the main goals was to keep the new language small, simple and efficient, and because of this some advanced functional languages features like lazy evaluation where not implemented.

The language was developed in parallel with a first product, a telephony system, which introduced in *Erlang* some pragmatism, so only the features really needed in practice where introduced.

The result was a process-based high level language, where the communication can only be done by asynchronous message passing. Programs written in *Erlang* are very easy to distribute, even if they were originally designed and implemented for running in only one processor. As in other declarative languages, variables cannot be rewritten, eliminating a big amount of side-effects in the sequential part of the language.

The language has been designed so that the cost of creating a new process is minimum. The programmer can create as many processes as needed without facing efficiency problems. The memory is garbage collected when it is not going to be used anymore.

Erlang is not as fast as lower level languages like C, but it has a very good performance in contexts that are adapted to its characteristics. In case some parts of the final solution need to be implemented using C, *Erlang* offers good interfaces for doing this in a natural way. In fact, using *Erlang* for the control system and implementing some low level parts in faster languages is a very common design decision that has been used in several *Erlang* projects.

Part of *Erlang* is the OTP (Open Telecom Platform), a set of libraries that allow the implementation of software without starting from scratch and having to reinvent the wheel. We will talk about this in the next sections, but the fact of providing a mature and complete platform and not only the core language is one of the keys of the *Erlang* success.

The main *Erlang* features are the following ones:

- Light-weight concurrency. An *Erlang* system can have thousands of different processes running and exchanging messages in the same machine. The language is designed for that and creating a process or sending a message are *cheap* operations. Therefore, when a new task needs to be carried out, the

natural solution in *Erlang* is to create a new process for it and synchronize it with the original one using message passing.

- Natural distribution. The language fits very well into developments that model problems where concurrency and distribution are present. Very sequential problems that could only be parallelized in a artificial way for performance reasons would not take full advantage of the *Erlang* capabilities.

- Built-in fault-tolerance. *Erlang* was constructed assuming that software fails. Although the language includes solutions for minimizing the possibility of including failures in the source code, and tools for checking that, the software written in *Erlang*, as happens with any language, will eventually fail. Thus, the true solution for making fault tolerant system is not removing failures but providing clean and built-in mechanisms for reacting whenever a failure (software or hardware) occurs. *Erlang* provides tools like linking processes, crashing messages and supervision trees that can be used in order to create a supervision structure on top of any application that will know what to do in the presence of an error (notify some processes, restart some others, reboot some components, etc.).

- Declarative functional syntax. Initially inherited from Prolog, and later converted into a key of the language, the syntax of *Erlang* has the simplicity and power of the high level functional languages. In the sequential part, side-effects are minimized and *everything* is a function. Concepts like higher order functions or list comprehension are completely supported by the language.

- Hot-code loading. For systems that run during years without stopping, the capacity of upgrading the software without needing to interrupt the tasks that are being carrying out is a must. *Erlang* provides mechanisms for upgrading the system while working, and even for using different versions of the same code in different parts of the same running system.

- Portability. Due to its pragmatic and industrially oriented history, every *Erlang* version supports different operating systems. As a consequence of this portability, for example, programs developed using *GNU/Linux* can easily be run on other UNIX operating systems. Also, advanced techniques used in the latest processors are supported, e.g. SMP (Symmetric Multiprocessing).

- Soft real-time. Hard real-time systems are those that can suffer failures if the very strict time constraints described by the programmer are not satisfied. This kind of software is not well supported in *Erlang*. However, the language behaves perfectly when the response times are of the order of milliseconds, but there is some flexibility with those numbers (soft real-time). Telecom systems are normally soft real-time.

- Interfaces with other technologies. *Erlang* is very good for distributed control subsystems, but for other kind of tasks, normally it is interesting to interact with external technologies. The language provides interfaces for communicating with C or Java, and also supports standards like Corba. Normally the external technologies are seen by the programmer from the *Erlang* side

as special processes, and communication with them is performed also using message passing.

- Complete development platform and design principles. Years of developing industrial problems have produced, together with the language, a big number of mature libraries implementing common problems, and a set of design principles that a developer should follow when writing *Erlang* software. Design principles include implementation patterns, that in *Erlang* are called *behaviours*, and offer a very clean way of reusing source code.

Further information on the *Erlang* philosophy can be found in [Arm03, AVWW96].

3.1.3 Sequential *Erlang*

The main data types that can be used in *Erlang* are integers, floats, atoms, tuples, lists, records and `funs`. Atoms are special *constants* that are frequently used inside complex data terms and always start with a lower-case letter, as opposed to variables, that start with upper-case. Examples of tuples are {`1,2,3`} (tuple with three integers), {`X,Y`} (tuple with two variables), or {`x,X,12`}, the tuple formed by one atom, one variable and one integer. Examples of lists are `[1,2,3]`, `[X,Y]` and also `[x,Y,12]`; lists can contain heterogeneous data types. `A++B` can be used for concatenating two lists `A` and `B`. `[a|B]` is used for creating a new list with the element `A` as the head and the list `B` as the tail. Records are special kinds of tuples where each of the elements can be accessed using a pre-declared name. A record can be declared as `-record(person,{name, surname, age=20})` (the third field has a default value), created as `Person = #person{name=''John'', surname = ''Locke''}`, and one of the elements is accessed evaluating `Person#person.name`. Finally, `funs` are anonymous functions that can be used temporarily and passed as parameters to other functions.

The *Erlang* source code is structured in modules. A very simple example of an *Erlang* module implementing the factorial is shown below:

```
-module(factorial).
-export([fact/1]).

fact(0) ->
    1;
fact(N) ->
    N * fact(N-1).
```

The first line declares the name of the module, which should be also the filename. The second line is a list of functions that are exported from the module. Only the functions that are in that list can be called from outside the module. Finally, the two clauses that implement the factorial are declared. Pattern matching is used in order to select which clause should be evaluated. The way of accessing the function from outside the `factorial` module would be, for example, `factorial:fact(10)`.

Erlang provides also conditional evaluation commands, namely `case` and `if`, and also *guards*, which are conditions that can be added to any function clause in order to complement the pattern matching.

List comprehensions provide syntactic sugar for creating lists by combining other lists. `[X,Y || X <- A, Y <- [1,2,3]]` creates the list with all the possible pairs where the first element is a member of the list `A` and the second element is 1, 2 or 3 (one element of the second list provided). Besides, pattern matching can be used in the left side of the list comprehension, and extra conditions can be added to the right side of the expression, in order to decide which elements are generated.

A sequential *Erlang* program is a set of modules that call each other (using the exported functions) in order to resolve a given task.

Standard libraries for manipulating tuples or lists are provided, including the typical higher order functions like `map`, `fold`, `foreach`, etc.

Detailed extra documentation on the sequential part of the language can be found in the *Erlang* book [AVWW96].

3.1.4 Concurrent and distributed *Erlang*

What makes *Erlang* quite different from other declarative languages are its concurrent and distributed features.

A special function `spawn` can be called inside any function in order to create a new process. The simplest version of that function is `spawn(Mod,Fun,Args)`. Whenever it is evaluated, a new process is created, which will start evaluating `Mod:Fun` with the arguments specified in `Args`. The function returns the *process identifier* and the code of the original process keeps executing in parallel with the new one.

Using the expression `Pid!Msg` a message `Msg` is sent from the current process to the one with the process identifier `Pid`. The message can be any *Erlang* term, and sending a message is asynchronous and never fails (i.e., if it is sent to a non existing process it is just silently discarded): the message is always stored in the mailbox of the receiver.

In order to receive a message, the following command is used:

```
receive
    Pattern1 -> Expr11,..., Expr1N;
    ...
    PatternM -> ExprM1,...
    after Timeout -> TExpr1,...
end
```

When a `receive` command is reached, the process looks into its mailbox for the first message matching any of the patterns. If it is the case, the list of expressions on the right side is evaluated and the execution continues; if no message matches, the process evaluation is suspended temporarily waiting for another incoming message. If no message has been received after `Timeout` milliseconds, the execution is continued anyway (the waiting time is infinity by default).

An example of a very simple echo server is shown below:

```
-module(ping).

-export([start/1, init/2]).

start(Times) ->
```

```
    spawn(?MODULE, init, [Times]).

init(Times) ->
    loop(Times).

loop(0) ->
    ok;
loop(Times) ->
     receive
{Pid, MsgContent} ->
    Pid!{MsgContent, Times-1},
            loop(Times-1)
    end.
```

The `start` function is used in order to create a new process that is going to work as the echo server. The new process starts evaluating the `init` function, which in this case does not do anything more than starting the server loop (in other cases the init function is used and therefore it was left here for didactic purposes) with one argument. `Times` represents the number of times that the server returns a message before it normally stops (when the loop finishes and the process does not have more code to evaluate, it is stopped and the memory associated is freed by the garbage collector). Whenever a message containing the process identifier together with the content is received, it is sent back to that process together with the number of loop iterations that are left before the echo server finishes.

Erlang provides mechanisms for registering a process giving it a name that can substitute the process identifier.

Two *Erlang* processes can be bi-directionally linked. When a process terminates abnormally (due to any kind of error), a signal is sent to all the rest of the processes that are linked to it. On receiving that signal, the default behaviour of the receiving process is to crash abnormally also. The crashing-in-chain behaviour can be stopped by configuring a process for trapping the exit signals: the signal would be converted into a crashing notification message that is stored in the receiver mailbox. The error trapping mechanisms allow the implementation of complex supervision structures (e.g., supervision trees).

Distribution is added to *Erlang* almost transparently. Instead of the basic `spawn` function, there is another one where the programmer can specify which node (machine running an *Erlang* node) should the new process be created in. Instead of just the process identifier, the *Erlang* node needs also to be specified for sending messages. The rest of the code is going to stay exactly the same, which makes very easy to distribute programs initially designed for just one machine.

Normally some of the constructs and commands we have seen are not directly used in the industrial *Erlang* code, because it is not built from scratch but on top of *Erlang* behaviours. This components, e.g. the generic server, hide the low level details for registering, sending and receiving messages. For example, the function `gen_server:call(Server, Message)` does internally the sending of the message and also the receiving of the answer, providing that answer as the return value of the function.

3.1.5 Open Telecom Platform

The set of libraries, applications, utilities and modules that are released together with each new *Erlang* version, are grouped under the name Open Telecom Platform (OTP). This is why the term *Erlang/OTP* is sometimes used to refer to the whole platform.

As it is usually the case in this kind of languages, *Erlang* is formed by a kernel, on top of which the libraries are run. The kernel is normally called Erlang Runtime System, and includes the virtual machine, the kernel itself, and the standard libraries. The standard libraries include modules that allow the manipulation of the main *Erlang* data structures. There are modules for handling lists, strings, files, calendars, input/output, more complex structures like directed graphs, or sets managed as ordered lists.

Erlang behaviours are also included as part of the standard library. The behaviours are implementation patterns that abstract some code functionality frequently used in a *Erlang* system. An example of this is the code of a server process: every server is similar, the only differences with other servers are the concrete messages it can receive, how it updates its internal state, and which answers are sent back to the caller. Creating a behaviour for a server is done creating the code for an *Erlang* module that implements all the basic functionality of a server and delegates in a call-back module, provided by the developer, when it needs to receive a message, to update the state, or to send back and answer. This way, the behaviours are generic code that can be parameterized by the developer in order to configure it for the concrete problem that needs to be solved.

Apart from the design principles and the *Erlang behaviours*, OTP includes a complete set of general purpose libraries and applications. OTP is very well documented and it is not the purpose of this section to describe it in detail. Interesting applications include `Mnesia`, a distributed database that allows automatic and transparent replication of data and has a specific query language called Mnemosyne, that is based on the concept of list comprehension; and `Eva`, which uses `Mnesia` internally and is an application for alarms and events handling which also offers logging facilities. Examples of interesting libraries included inside OTP are `erl_interface`, which makes possible for *Erlang* processes to talk to C programs seeing them as normal processes which can receive messages; `GS`, a library that supports the creation of graphical user interfaces; `ASN.1`, which supports the specification of communication protocols and data structures independent from the language; or `SSL`, which has the functionality for creating secure communications using sockets.

For further information, the *Erlang* OTP documentation [Eri06] can be consulted.

3.1.6 *Erlang* success stories

Erlang is the distributed functional language more widely used in industry. Ericsson has used it for almost two decades now for developing all kind of embedded control systems. The best example of *Erlang* use inside Ericsson is the AXD 301 ATM switch, a high speed, modular and scalable switch that is used in some European countries for controlling the backbone of the national telephony systems.

The control system of the switch has about 1.7 million lines of *Erlang* source code, which control the hardware and only the low level modules are written in C and C++ code.

In the last decade, since *Erlang* went *Open Source*, new companies have started to appear. The first ones where spin-offs of Ericsson, devoted to develop fault tolerant highly concurrent systems, including massively parallel robust e-mail servers, web servers or LDA servers. They showed that *Erlang* was not only a very powerful language for telecom applications, but also for any kind of task-oriented, very parallel problem. In the Internet technology the strengths of the language fit very well.

New companies are appearing constantly, now outside Sweden and Ericsson, creating different kinds of products or offering support and consultancy for *Erlang*. Also some *Open Source* projects developed using *Erlang* have emerged, as it is the case of `ejabberd` one of the most know `jabber` protocol servers; `wings3d`, a 3D subdivision modeler; or `yaws`, a fast web server.

The *Erlang* community, and therefore the number of success stories, both companies and projects (both *Open Source* and proprietary ones), keeps growing at high speed.

3.2 Distributed software development and design patterns

Some concepts related to development methodologies and design patterns for distributed software development are used during the rest of the thesis. In this subsection those concepts are presented and described to the reader.

3.2.1 Design and implementation patterns

A *design pattern*, in the context of software engineering, is a general solution to a common design problem. Normally, the design patterns do not say how the actual code derived from the design is going to look like, they just depict how the different objects (or processes) interact in order to solve the problem, and this makes the pattern reusable in different contexts.

Although they are nowadays widely used in software engineering, the original idea of the *patterns* came from a different field: civil engineering. In 1977, the very influential book *A Pattern Language: Towns, Buildings, Construction* [AIS77] was published. The book described several hundreds architectural patterns, and insisted in the importance of reusable solutions for well known problems in architecture. About ten years later, the first experiments [BC87] and discussions about applying these ideas to software engineering were started. In 1994, the book *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV94] was published, and the use of design patterns gained a lot of popularity.

Design pattern descriptions have a well-known structure that documents when they should be used, the actors that participate, and how it solves a recurrent problem (what is normally called a micro-architecture, including classes, objects and methods that participate). Aspects like the consequences of using the pattern, related patterns or even sample source code can also be included.

Besides helping in finding solutions to problems without needing to reinvent the wheel each time they appear, patterns are also very useful as a standard language for making the communication between architects, designers and developers easier.

Nowadays, design patterns are very popular in the object oriented world, and variations or specializations of the concept have originated related ideas like *architectural patterns* or *implementation patterns*. The difference among them is mainly the level of abstraction of the proposed reusable solution, and therefore the point in the development process where they should be used. Architectural patterns are very general high level architectural solutions about how the components of a system interact, and implementation patterns are low level ways of reusing code, or abstracting some general behaviour in order to structure the program correctly, making it easier to maintain.

The original concept of *design pattern* fits very well in the object oriented programming and most of the related work has been carried out in that field. However, the idea can be adapted and extended to other philosophies like the distributed process-based approach of languages like *Erlang*.

3.2.2 *Erlang behaviours*

As we have already introduced in the previous section, the *Erlang* platform includes a set of *behaviours*, concepts similar to the algoritmic skeletons [Col89], which can be seen as a combination of ideas from the original design patterns mixed with the concept of implementation patterns.

A behaviour implements the generic *Erlang* source code needed for solving a recurrent problem into a special parameterizable module. This code is widely used and therefore very well tested, and it can be trusted by the developers, who only need to provide what it is called the call-back module: a set of functions that parameterizes the generic code in order to do exactly what it is needed in each concrete scenario. Although the behaviours are basically source code, they force the developers using them to structure the code in a concrete way, and sometimes even to structure the processes of the system following a fixed pattern. This is why we said that they provide design and implementation level patterns.

The *behaviours* included in *Erlang/OTP* are:

- `application`: defines a general framework for implementing *Erlang* applications as components that can be easily started, stopped and reused. If the developer wants to use this pattern, an application file describing the modules that compose the application and the name of the call-back module needs to be provided. In the call-back module the developer can specify the code that should be evaluated when the application is started or stopped.

- `supervisor`: a supervisor is a special process that has the responsibility of starting, stopping and monitoring a set of processes. The programmers using this pattern should specify the list of child processes (child specifications), which are going to be started in the order specified in that list. As part of the specification for each of the children, the restarting policies are decided by the developer. This pattern permits the creation of a tree-like structure of supervision, one of the techniques used by *Erlang* in order to create fault tolerant

systems. *Erlang* offers another complementary pattern called `sup_bridge`, which connects a process to a supervision tree even if it is not designed with that philosophy in mind.

- `gen_event`: allows the easy programming of event handling and logging mechanisms. In the OTP platform, an *event manager* is a special process to which the system events are sent. Each event manager has a list of associated *event handlers*. When a new event is received in the event manager, it is processed by each of the event handlers. A trivial example of an event handler is an event logger that writes to a file all the events received. Each event handler is implemented as a call-back module for the event manager process. The call-back is going to be called with its state (kept by the manager) and the new event that has been received. This is an example of classical design pattern (*observer*) implemented using *Erlang* behaviours.

- `gen_server`: used for easily creating client/server programs, it implements the generic functionality of a server, and can be parameterized as described above. The developer using the pattern provides a call-back module including: the function for initializing the state of the server and the functions for receiving messages and sending answers back after updating the internal state. We will explain this pattern in detail in the next section.

- `gen_fsm`: similar to the generic server, but allows the creation of servers whose answer and new server state depends on a finite state machine. Two notions of state coexist in the `gen_fsm`: the server state and the finite state machine state. Therefore, the developer provides a function for initializing the state, but instead of providing a unique function for receiving messages and computing the answer and the new server state, a set of functions are provided. The function that is going to be called depends on the current state of the FSM. A generic FSM can be easily implemented using a generic server pattern, but the resulting code is cleaner if the specialized pattern is used.

Some extensions of the basic *Erlang* behaviours have been already proposed. In [Eks00], a set of design patterns for simulation software are described and implemented as behaviours. In the Chapter 5 of this manuscript, we will describe how *VoDKA* also makes advanced use of the patterns provided by *Erlang* and extends them adding interesting design and implementation patterns useful for *VoD* servers.

3.2.3 An example *Erlang* behaviour: the generic server

In this section we describe how the call-back modules for an *Erlang* generic server, that are going to be seen several times throughout the thesis, are implemented.

The structure of a basic call-back module would be as specified in the following pseudo-code:

```
-module(modulename).
-behaviour(gen_server).
```

```
-export([start_link/0]).
-export([init/1, handle_call/3, handle_cast/2]).

start_link() ->
    gen_server:start_link(ServerRegName, ?MODULE,
                          InitArgList, GenServerOptions).

init(InitArgList) ->
    ...
    {ok, InitialState}.

handle_call(MessagePattern1, From, State) ->
    ...
    {reply, ReplyMessage1, NewState1};
...
handle_call(MessagePattern2, From, State) ->
    ...
    {reply, ReplyMessage2, NewState2}.


handle_cast(MessagePattern3, State) ->
    ...
    {noreply, NewState3};
...

handle_info({'EXIT', Pid, Reason}, State) ->
    ...
    {noreply, State1}.
...
```

The first two lines declare the name of the call-back module and specify that it is in fact an implementation of a generic server. This is used by the compiler in order to check if all the mandatory functions that a call-back module must have are actually implemented. After that, the list of exported functions is provided: all of them need to be exported because they will be called from the generic server and from other modules.

The `start_link` function is used for initializing the generic server. The four arguments specify the name for registering the server in the node, the name of the call-back module (specified in this case by an *Erlang* macro that is converted into the current module filename), the list of arguments that are going to be used for evaluating the function `init`, and a list of generic server options (timeout for the initialization, debugging options, or spawn options). When the server is initialized, it registers the process, calls to the `init` function for computing the initial state, and then waits for incoming messages.

The functions `handle_call`, `handle_cast` and `handle_info` are going to be called by the generic server process whenever one message is received. The first one is used for synchronous messages, those that are going to expect an answer. The second one is used for asynchronous communication. Finally, the last function is used when any kind of special information related to the crashing of other processes is received.

External processes can use the following two expressions for sending a message to a generic server:

```
gen_server:call(ServerName,Message,Timeout)
gen_server:cast(ServerName,Message)
```

The first expression makes a synchronous call to the generic server identified by `ServerName`. The `Message`, which can be any *Erlang* term, is sent to the process, and `Timeout` is the maximum number of milliseconds that the answer can be delayed. The generic server receives the message and passes it as argument to the function `handle_call` of the call-back module. A new state is then computed and the answer message is sent back to the caller process. The implementation of the `call` function of the generic server receives that message and returns it as a value.

The second expression makes an asynchronous call to the generic server identified by `ServerName`. The `Message` is again sent to that process, but no answer is expected; the `cast` function ends immediately. The generic server receives the message and forwards it as argument to the `handle_cast` function of the call-back module, which is only going to update the internal state without sending a message back.

Besides the functions `call`, `cast` and `start_link`, the generic server module also implements other functions for performing multicalls, starting the process without linking, or for explicitly replying a value to a process without using the *handle* functions.

Also, the provided call-back module should also have two extra functions: `terminate`, which does the clean up that is necessary just before the generic server is finalized; and `code_change`, which is used for the generic server in order to update its internal state without stopping while a code upgrade is being done.

3.3 Process algebras

The methods and tools described in this thesis are heavily based on the concept of process algebra. In this section, we start discussing what process algebras are in general, and we finish describing in detail μ`CRL`, a process algebra with data that is going to be used in the second part of this thesis.

3.3.1 Introduction to process algebras

Process algebras are mathematical formalisms for modelling concurrent systems. The history of process algebras [Bae05] starts in the seventies. At some point, operational semantics, denotational semantics and axiomatic semantics turned out not to be easy to adapt to giving semantics to programs using some kind of parallel operator.

Process algebras allow the description of interaction, communication and synchronization between processes in a formal way. This means that they respect algebraic laws that allow the descriptions to be analyzed, formally reasoning about them, e.g. similarity between different processes can be studied using bisimulation equivalence.

The features that are normally cited as shared by all the existing process algebras are [Pie97]: they formalize message-passing interactions between processes without using shared memory; they describe the processes behaviour using a small set of primitives and operators; and they define algebraic laws for the operators, making possible the usage of equational reasoning with the process algebra specifications.

The following operators are normally present in every process algebra:

- Parallel composition of two processes: $P|Q$. It means that the processes P and Q are executed in parallel. Depending on the concrete process algebra, an asynchronous or synchronous channel is created in order to permit the exchange of messages between processes.

- Communication between processes. It is normally based on two primitives, one representing the sending of a message ($x\langle y\rangle$, where x is a synchronization expression and v is the message sent), and the other representing the reception ($x(v)$, where x is the same synchronization expression and y is potentially a pattern matching-like expression that can be used in order to bound variables with the *received* information).

- Sequential composition of two commands: $x \cdot y$. It means that the operator x should be executed before executing y. An example of the sequential composition would be $x(v) \cdot P$, meaning that before executing P, a message matching v should be received over the channel x.

The reduction semantics of synchronous communication between two processes can be summarized in the following rule:

$$x\langle y\rangle \cdot P \mid x(v) \cdot Q \longrightarrow P \mid Q[{}^{y}/_{v}]$$

The left side of the expression represents the parallel execution of two processes, one first sending a message and then executing P, and the other first receiving a message and then executing Q. In that configuration, a synchronization with message-passing can take place, giving rise to the situation represented by the right side of the expression: P is now being executed in parallel with Q, where all the free variables in v are replaced by the values bounded after pattern matching with y.

Apart from the basic operators, process algebras normally include also hiding capacities (allowing a possible interaction to be avoided when it is not desired), implementation of recursive expressions (creating potentially infinite computation), and other aspects like conditional execution or non-determinism (it can already be implemented using parallel composition and sequential composition, but some process algebras implement a specific operator for this in order to simplify the specifications).

Classical examples of process algebras are CSP (Communicating Sequential Processes) [Hoa78], CCS (Calculus of Communicating Systems) [Mil82] or ACP (Algebra of Communicating Processes) [BT85].

More recently, evolutions of the classical process algebras have been created, e.g. π-calculus [Mil99] is an extension of CCS including more advanced concepts

like mobility; extensions for including time notions inside process specifications have been also proposed.

More information on process algebras can be found in [Mil95, Hoa85, Fok00].

3.3.2 μCRL: a process algebra with data

μCRL [Gro97] (micro Common Representation Language) is a specification language based on the concepts of the process algebras, developed in the CWI (Centrum voor Wiskunde en Informatica) of Amsterdam.

μCRL is intended for the description of real processes where data is relevant. The language was designed with three main objectives in mind: being applicable to complex distributed systems, being simple enough for allowing complex mathematical analysis, and being precise so that a set of tools can be constructed for automating the usage of the language in real projects. It is defined by its creators as: *a process algebraic language that was especially developed to take account of data in the study of communicating processes. It is basically intended to study description and analysis techniques for (large) distributed systems.*

As opposed to classical process algebras, where the introduction of complex data has to be done using infinite sets of equations where variables are used for including data values, in μCRL data is defined using equational abstract data types (besides, conditional operations and `sum` over a data type are included).

A μCRL specification is composed by two main parts: the specification of the data types managed by the processes, and the specification of the processes interacting in the system.

The data part involves five main keywords:

- `sort`: sorts representing a non-empty set of data.
- `func`: functions that act as basic constructors of a given sort.
- `map`: declaration of the operations on the elements of a sort.
- `var`: variables needed for the rewriting rules.
- `rew`: equations that represent the existent properties and relations on the sort operations.

For example, the first part of the definition of the sort for the natural numbers can be seen below:

```
sort
     Natural
func
     0: -> Natural
     s: Natural -> Natural
map
     plus,times,minus,rem,div: Natural # Natural -> Natural

     eq,leq,less: Natural # Natural -> Bool

     if: Bool # Natural # Natural -> Natural
```

Two constructors for producing terms of the sort `Natural` are defined, one for the zero (without arguments) and other for the successor (receives another natural number). E.g., `s(s(s(0)))` is therefore the representation for the number 3.

Three sets of operations are declared: arithmetic operations over naturals; relations between numbers; and conditional evaluation for naturals. The sort `Bool` is assumed to be declared in other part of the μ`CRL` specification.

In order to define the rewriting rules for all those operations over the sort, three `Natural` variables need to be defined. The code of the rewriting rules would be as follows:

```
var
     N,N1,N2: Natural
rew
     plus(0,N) = N
     plus(s(N1),N2) = s(plus(N1,N2))
     minus(N1,0) = N1
     minus(s(N1),s(N2)) = minus(N1,N2)
     times(0,N) = 0
     times(s(N1),N2) = plus(N2,times(N1,N2))
     rem(N1,N2) = if(less(N1,N2),N1,rem(minus(N1,N2),N2))
     div(N1,N2) = if(less(N2,N1),s(div(minus(N1,N2),N2)),0)

     eq(0,0) = T
     eq(0,s(N2)) = F
     eq(s(N1),0) = F
     eq(s(N1),s(N2)) = eq(N1,N2)
     leq(0,N2) = T
     leq(s(N1),0) = F
     leq(s(N1),s(N2)) = leq(N1,N2)
     less(N1,0) = F
     less(0,s(N2)) = T

     if(T,N1,N2) = N1
     if(F,N1,N2) = N2
```

Processes are formed by sequences of activities called *actions*. Each process is defined by a process term. Data (and the associated rewriting rules) takes part in the processes as parameter for processes and actions, therefore actions have associated types, called *domains*, that represent which data they are going to handle.

Continuing with our previous example, the declaration of two very simple actions for sending and receiving one natural numbers and one boolean would be as follows:

```
act
  snd_update: Natural # Bool
  rcv_update: Natural # Bool

comm
  snd_update | rcv_update = updatesent
```

The `comm` part indicates that the actions `snd_update` and `rcv_update` can synchronize, producing the communication action `updatesent`.

Besides the user defined actions, there are two predefined ones that are already available in the language: `delta`, a deadlock action that never synchronizes, and `tau`, an internal action, that denotes any activity in the system that cannot be observed.

Processes are defined using μCRL *process terms*, where the order in which the actions can take place is declared. Each process consists of basic process terms combined using operators. The following specification describes a process that receives a pair with a number and a boolean, and if the boolean is true, it adds the number to its internal state, and if not, it subtracts it:

```
proc UpdateNumber (n: Natural) =
   sum(X:Natural,
     sum(B:Bool,
        rcv_update(X,B).
        (UpdateNumber(plus(n,X))
         <|equal(B,T)|>
         UpdateNumber(minus(n,X)))))
```

The `UpdateNumber` process is recursive and has a state (parameter) that is a natural number. The `sum` expression means that the variables `X`, of sort `Natural`, and `B`, of sort `Bool`, can take any value when used in the rest of the process. This is needed because the action `rcv_update` is required to be able to synchronize with any possible number and boolean value that is going to be received. When another process executes the action `snd_update`, the pattern matching is evaluated and both variables in the action are bound to the concrete values.

The $\cdot$ sequence operator is used for separating the first action and the second part of the process. In that second part of this example process, the conditional evaluation operator of μCRL is used. In case the value of `B` is `T` (the true value in the sort `Bool`), the sentence above the comparison is executed, adding `X` to the process state; in other case, the value of `X` is subtracted.

A process activating this one could have the following specification:

```
proc SendNumber =
   (rcv_update(s(0),F) + snd_update(s(s(0)),T).
   SendNumber
```

The process is again recursive and has now arguments.

The `+` operator represents the non-deterministic choice in μCRL. This simple process will iterate forever, and in each iteration either the number `1` will be sent for subtracting from the state of the remote process, or the number `2` will be sent for addition.

The last part of any μCRL specification, once all the data types, the actions, and the processes are declared, is the initialization, where the processes are started (called the *initial behaviour* of the system). For our simple example, the following `init` part would be needed:

```
init
   SendNumber() || UpdateNumber(s(s(0)))
```

The `||` operator represents the parallel composition of processes in the specification language. The first process has no parameters and the second one is initialized with the value 2.

μCRL *process terms* are really like functions in a distributed functional language, and can be called from other process terms. It is the `init` part of the specification what creates the real processes (the threads of execution), that are going to be executed in parallel and can go through different process terms.

The fact that two actions can synchronize does not mean that they need to do it. Actually, if a communication of the form $a|b = c$ is defined, and the current execution is $a||b$, this is exactly equivalent to $a.b + b.a + c$, i.e., only one out of the three possible cases represents the actual synchronization.

Three extra operators can be used in the `init` part: `encap`, `hide` and `rename`.

`encap` is used when we want to force the communication to take place, avoiding individual actions: `encap({a,b},a||b)` is equivalent to c (a and b cannot happen without synchronization anymore).

`rename` is used when we have several processes with the same structure but we want to modify the way each of them is going to synchronize with the rest of the system. `rename(a1->a2, b1->b2, P)` substitutes the action $a1$ by $a2$ and the action $b1$ by $b2$ inside the process P.

Finally, `hide` is used to reduce the complexity of the externally observed behaviour of a system. Sometimes a subset of the system actions are not relevant for an analysis of the system, and they can be converted to the invisible *tau* action. `hide({a,b},P)` converts the actions a and b into *tau* inside the process P.

Below, a more complex example is shown. The process implements a message queue:

```
act
  recv_from_queue, send_from_queue, remove_from_queue: MessageSort
  send_to_queue, recv_in_queue, add_to_queue: MessageSort
  notify_queuefull

comm
  recv_from_queue | send_from_queue = remove_from_queue
  send_to_queue | recv_in_queue = add_to_queue

proc
  Queue(Messages: QueueContent) =
     (notify_queuefull.
       send_from_queue(gethead(Messages)).
        Queue(rmhead(Messages)))
     <| maxqueuesize(Messages) |>
     (sum(Msg: MessageSort,
          recv_in_queue(Msg).
              Queue(add(Msg,Messages)))) +
     (send_from_queue(gethead(Messages)).
              Queue(rmhead(Message)))
```

The queue receives messages from other processes that have evaluated the `send_to_queue` action, and it stores them in the internal state until another process reads from the queue using pattern matching over the action `recv_from_queue`.

If the maximum size allowed for a queue in the system has been reached (`maxqueuesize` returns true), the upper part of the conditional operator is executed. That part is formed by a sequence executing: first an action that notifies the fact of the queue being full, then a communication action that is going to check if someone wants to remove a message from the queue, and finally -once that synchronization takes place- the recursive call that starts again the process after removing the head of the queue from its state.

If the queue is not full, and new messages can still be received, the bottom part of the conditional operator is executed. That part is a non-deterministic choice between adding any new received message to the queue or removing the head of the queue. In this latter case, the head of the queue is compared by using pattern matching with the parameter of the action `recv_from_queue`, used in the external process that is accessing the queue.

The rewriting rules `add`, `rmhead` and `gethead` operate over the sort named `QueueContent`.

An extended version of this queue implementation is used later in the thesis in order to model in μCRL the buffer of an *Erlang* generic server process. As we have seen in the previous sections of this chapter, the generic server can receive three kinds of messages: `call`, `cast` and `info`. The μCRL specification of the extended buffer process is shown below:

```
act
  bufferfull: Term
  gen_server_call,gscall,buffercall: Term # Term # Term
  gen_server_cast,gscast,buffercast: Term # Term
  send,gsinfo,bufferinfo: Term # Term
  gshcall,handle_call,call: Term # Term # Term
  gshcast,handle_cast,cast: Term # Term
  gshinfo,handle_info,info: Term # Term

comm
  gen_server_call | gscall = buffercall
  gen_server_cast | gscast = buffercast
  send | gsinfo = bufferinfo
  gshcall | handle_call = call
  gshcast | handle_cast = cast
  gshinfo | handle_info = info

proc
    Server_Buffer(MCRLSelf: Term, Messages: GSBuffer) =
     (bufferfull(MCRLSelf).
      (gshcast(MCRLSelf,cast_term(Messages)).
         Server_Buffer(MCRLSelf,rmhead(Messages))
       <| is_cast(Messages) |>
       (gshinfo(MCRLSelf,info_term(Messages)).
          Server_Buffer(MCRLSelf,rmhead(Messages))
        <| is_info(Messages) |>
        (gshcall(MCRLSelf,call_term(Messages),call_pid(Messages)).
           Server_Buffer(MCRLSelf,rmhead(Messages))
         <| is_call(Messages) |>
         delta())))
```

```
<| maxbuffer(Messages) |>
(sum(Msg: Term,
     sum(From: Term,
         gscall(MCRLSelf, Msg, From).
         Server_Buffer(MCRLSelf, addcall(Msg,From,Messages)))) +
 sum(Msg: Term,
     gscast(MCRLSelf, Msg).
     Server_Buffer(MCRLSelf, addcast(Msg,Messages))) +
 sum(Msg: Term,
     gsinfo(MCRLSelf, Msg).
     Server_Buffer(MCRLSelf, addinfo(Msg,Messages))) +
 (gshcast(MCRLSelf,cast_term(Messages)).
    Server_Buffer(MCRLSelf,rmhead(Messages))
  <| is_cast(Messages) |>
  (gshinfo(MCRLSelf,info_term(Messages)).
     Server_Buffer(MCRLSelf,rmhead(Messages))
   <| is_info(Messages) |>
   (gshcall(MCRLSelf,call_term(Messages),call_pid(Messages)).
      Server_Buffer(MCRLSelf,rmhead(Messages))
    <| is_call(Messages) |>
    delta))))
```

The implementation is almost the same than before, but now for each of the previous actions we have three (one action for each kind of message), and the code is also triplicated so that all the different kinds of messages are added, removed and pattern matched correctly from the queue.

3.4 Formal verification of distributed systems: state of the art

In this section, the state of the art in the formal verification of software is sketched. The section talks about existing tools and techniques for the verification of concurrent systems, specially for those that are developed using functional languages. The main content of the section is a summary of [Só1], which was already an extension of *Formal Methods: State of the Art and Future Directions* [CW96], the very influential paper written by Clarke and Wing in 1996.

The term *Formal Methods* refers to the application of mathematical methods to the specification, analysis, design, implementation and maintenance of software and hardware, i.e. to the application of formal tools and techniques at any point of the system development process.

In the latest years, the use of formal methods for verification purposes has been quite popular in hardware verification [Fis96] and also in the verification of communication protocols [HS96]. Due to the complexity of real software, the application of the same kind of tools and techniques to software is more challenging and has followed a different pace.

The more common methods for system verification are *model checking* (the exploration of the whole state space diagram to see if a property holds), and *theorem proving* (formally reasoning step by step about system properties). The first method has some problems related to the state space explosion and the impossibility of working with infinite structures (they would generate infinite paths). The

second one is, in principle, more powerful and generally applicable, but it is also more complex and it needs more human interaction during the verification process.

In both cases the final goal of the verification is to check if the system satisfies a set of properties, that are expressed in some kind of formal logic (e.g. μ-Calculus, CTL, LTL, XTL).

Despite of being very heterogeneous, formal verification techniques normally combine three main components [HR00]: (1) a framework for system modelling, normally a specification language; (2) a property specification language, normally a temporal logic; and (3) a verification method that supports the checking of the properties in the model of the system.

Using that common structure, a classification of the different approaches can be done based on the following five parameters [HR00]:

- Checking procedure used: model checking or theorem proving.
- Automation level: from completely automatic methods to those where each step requires human interaction.
- Specification level: from a high level simple description of a system to a very detailed specification where all the low level is included (almost as detailed as the programming language). The required level of detail normally depends on the property to be verified.
- Application domain: some techniques are very generic but others are specialized (hardware or software, concurrent systems and sequential systems, reactive systems or input/output ones).
- Moment in the development process when it is used: different techniques can be used in different phases of the development.

For the description of the state of the art, enumerating languages, techniques and tools, we will use the classification presented in [CW96]: specification languages (both for systems and properties); model checking techniques and tools; and theorem proving techniques and tools.

3.4.1 Specification languages

They are mathematical languages with clearly defined syntax and semantics, used for formally describing a system, or the properties we want to verify about a system.

Two of the classical languages for specifying sequential systems, where states are rich structures and transitions are normally represented by preconditions and postconditions, are Z [WD96] and VDM [Jon90], both created in the late 80s.

Also during that decade the most relevant languages for describing concurrent systems were created. Some of them are process algebras and have been introduced in Section 3.3, but others are based on different kinds of formalisms, e.g. Statecharts [Har87] and Temporal Logic [MP92]. Normally, in the concurrent specification languages, states lose importance and are much simpler, while the modelling is focused in the structure, order and relations between events during the system execution.

There are some mixed specification languages, where richer states can be described in combination with the concurrency model, as it is the case of RAISE [Gro93], LOTOS [BB87] or μCRL, already explained in detail in Section 3.3.

These specification languages are usually closer to the programming languages. In a level of abstraction closer to analysis and design, UML (Unified Modelling Language) [RJB04] can be considered an specification language. The same happens with SDL [EHS97] for the description of process-based concurrent systems.

3.4.2 Model checking

Following the classification we have presented, the verification using model checking is model based, automatic, can be used for concurrent and reactive systems, and it can be used in different phases of the software development. This technique is based on the construction of a formal model of the system for later checking if a set of properties hold for that system. In order to check that, all the possible execution paths need to be verified (thus, the model needs to be finite). The problem can still appear when the model, being finite, has a considerable size; for that reason, research efforts have been made for reducing the size of the generated state spaces.

The main line of use of this technique is temporal model checking [CE82], where the systems are modelled with state transition diagrams and properties are specified by using some kind of temporal logic. The notion of truth in temporal logics is dynamic: it depends on the state of the system where the formula is checked. In the other approach to model checking, both the system and the properties are modelled as state transition diagrams, and the checking of the property is done by comparing diagrams [FGKM96, CPS93a].

The main advantages of model checking are:

- High degree of automation and relatively fast answer.
- If the property does not hold for a system, a counterexample with one of the paths not satisfying the property is provided.
- It is easier to use in the first phases of the software development because of being less time-consuming.
- The learning curve is less steep.

The main disadvantages of the technique are:

- Limited to finite models. This problem can be solved sometimes with an abstraction: parts of the system details are hidden until the new state space is not infinite anymore. If the property holds for the reduced graph, we cannot say for sure what happens with the original one, but if it does not hold, we have also found a potential problem in the complete state space.
- State explosion. There are techniques for solving this problem, like Binary Decision Diagrams [Bry92], that represent more efficiently the state diagrams.
- The *proof object* is not returned, as it is the case with theorem provers, which return a concrete proof for the verification. In some cases, like automatic code generation from the proof object [BS95], this can be an important disadvantage.

The first model checkers based on temporal logic were EMC [CES86] and CAESAR [FGKM96], in the early 80s. The second one evolved until today originating `CADP`, which is used in this thesis as explained in Section 8.4.

The first tool incorporating BDDs (binary decision diagrams, efficient representation of Boolean functions used for optimizing the model checking) was SMV [McM92], the first one using partial order reduction was SPIN [GPVW95], both created in the mid 90s and still very popular nowadays.

Among the initiatives and tools that are based on the comparison of state transition diagrams, some of the most relevant ones are Cospan/Formalcheck [Kur98], FDR [Sca98] and Auto [RdS91]. Created also in the mid 90s, Concurrency Workbench [CPS93b] combines both kinds of model checking, verifying properties expressed in μ-calculus against systems modelled using CSS.

3.4.3 Theorem provers

Following the classification we have presented, the verification using theorem provers is proof based, needs human interaction, and is normally oriented to the verification of properties. This technique is usually more adapted to I/O software (where an input is received and processed for generating a given output) than to reactive systems (those systems that are constantly reacting to external signals).

In theorem proving techniques, both the system and the properties are described in some kind of mathematical logic. The logic needs to be defined inside the formal system, which has a set of axioms and a set of inference rules that allow new formulas to be derived from the axioms. Thus, the verification process consists on deriving the properties from the initial axioms using the inference rules.

The main advantages of the approach are:

- It can be applied to systems with infinite state spaces, as the state space is not explored.
- The proof object is obtained as a result of the verification process.

On the other hand, the main disadvantages are:

- As it is less automatic, the process can be slower.
- Human interaction makes possible the introduction of errors in the process.
- More complexity in the verification process, and therefore steeper learning curve.

Among the main tools and techniques can be used in the process of verifying using theorem proving, some of the most relevant ones are Nqthm Boyer-Moore [BM88], that later would evolve into the more industrial version ACL2 (*A Computational Logic for Applicative Common Lisp*) [KM96], Eves [CKM$^+$91] and RRL [KZ95], all of them semi-automated initiatives.

In the group of the *proof checkers*, which help in the process of reasoning by providing all kind of rules and heuristics, the most popular ones are Isabelle [NPW02] (an evolved version of HOL [GM93]), Coq [CCF$^+$] and LEGO [Pol94].

Other tools combine model checking capacities with theorem proving functionality. Examples of this kind are HSIS [ABC+94], STeP [MBB+99], VIS [RGA+96], and PVS [OSR95], most of them including some model checking modules that are able to verify properties expressed in μ-calculus. Another example of mixed initiative is Analytica [CZ92], constructed using the popular tool Matematica.

3.4.4 Formal verification of functional and concurrent systems

We have classified and referenced the main tools from the areas of model checking and theorem provers. Some of them, as we have seen, are specially designed for being used with concurrent processes. In this section we will talk about some initiatives addressed specially to either functional or concurrent systems, and we will finish describing all the existent initiatives for applying formal methods to the distributed functional language *Erlang*.

Due to their intrinsically formal nature, based on concepts like function or referential transparency, functional languages are in principle good candidates for applying formal methods to them. However, that was not always the case during the latest years, when frequent examples of usage of model checking techniques, for example, were applied to Java or C/C++.

As a curiosity, the tool for verifying hardware called Forte [SJO+05] uses a lazy variation of the language Caml for the description of system properties, that are verified using model checking.

Sparkle [dMvEP01] is a theorem prover specially designed for verifying programs written using functional languages. The tool is written in Clean and is specialized in verifying programs written in that language.

PLogic [Kie02] is a logic for property verification of functional programs written in Haskell.

In [Yu98], a complete methodology for verifying concurrent systems based on type theory is presented. As integration platform, LEGO is used, and on top of it a specific interface and heuristics are provided, helping in the verification process. The system can be used for verifying either message passing concurrency, modelled using CSS, or shared memory concurrency, modelled using an *ad hoc* imperative concurrent language. Properties are written in μ-calculus, and other temporal logics like CTL or LTL are accepted. The framework includes also a model checking subsystem that allows the automatic implementation of the finite parts of the models.

The use of Coq for verifying properties of concurrent systems using theorem proving have been described in [AKY05] (the case study is a mail server written in Java) and [AK04] (a Coq library for helping in the verification of concurrent systems is proposed).

Other publications present different methodologies and techniques for the verification of concurrent systems: in [LGS+95] some conclusions on how to use abstractions for verifying concurrent systems are presented, and in [LL97] the verification of concurrent object-oriented systems is explored.

Combining both functional languages and concurrency, in [HS04], model checking techniques are used in order to verify LTL formulas for Concurrent Haskell properties at runtime.

3.4.4.1 Formal verification of *Erlang*

We have seen that *Erlang* is a distributed functional language with good features for developing control systems. Different initiatives have been started over the years for trying to reduce the presence of errors in *Erlang* programs.

Erlang is a dynamically y typed system. Some projects [SM97, AA98] for creating an static type system have been started with the goal of detecting more errors at compilation time, but the very flexible features of the language make it very difficult to introduce a type system without removing part of that flexibility.

In 1997, as part of an initiative funded by the Swedish government, the Computer Science Laboratory (CSLab) of Ericsson and the SICS (*Swedish Institute of Computer Science*) started a project with the goal of studying the use of formal verification for *Erlang*. The project had two parts, one for studying model checking techniques and the other for theorem proving.

The model checking study originated a translation from *Erlang* to the μ`CRL` process algebra (taking advantage of the *Erlang* behaviours), then the generation of the state space, and the verification of μ-`Calculus` properties against that state diagram. All the work carried out in this thesis is very related to those tools and techniques, and therefore they are described in detail in the following chapters.

A different alternative for doing model checking of *Erlang* software, based on abstractions from the point of view of the data and the control, is presented in [Huc99, Huc01] (in this case *Erlang* behaviours are not used).

The theorem proving study originated the creation of a new framework for reasoning about *Erlang* programs: EVT [DF98]. The first results [DFG98] included the verification of simple properties for a billing agent written in *Erlang*. After a new prototype, in 1999 the first studies [FG99, AD99] over real *Erlang* systems are carried out. In 2000, the focus was to increase the automation of the tool and even develop a graphical user interface for reducing the learning curve [ACD$^+$03]. EVT helps in the verification of properties described in μ-calculus, and supports parametric, inductive and co-inductive reasoning on the components and properties defined in a recursive way. The verified language is a subset of *Erlang* called *Core Erlang*, where distribution is not considered (only one node is modelled, in which many processes can be executed at the same time). ML, the language in which the tool is written, can also be used for writing high level tactics, that can be used for the automatic resolution of some parts of the verification process. The tool has been used also to verify a database lookup manager system [AD99] and a concurrent task management server [ACD$^+$03].

Other initiatives that have been more recently started for reducing the number of errors in *Erlang* programs are Dialyzer [LS04], a tool that does static analysis of *Erlang* modules in order to find errors like discrepancies, unreachable code, basic type errors or unsafe code; and QuickCheck [CH00, AH03], which generates automatically test cases for *Erlang* programs from formally described properties.

Part II

VoDKA development

VoDKA (*VoD* Kernel Architecture) is a research and development project that has been carried out during the past 6 years by a team from *LFCIA-MADS* (Models and Applications of Distributed Systems), a research group at the Computer Science Department of the *University of A Coruña*. The project started in January 2000, funded during the first two years by the European Union with regional ERDF funds. Over the next years new funds coming from the *Spanish Government*, the *Galician Government*, and the *University of A Coruña* were received. During these six years of research and development, the company *Cable y Telecomunicaciones de Galicia, S.A.* acted as industrial partner of *LFCIA-MADS*. Recently, the company LambdaStream, a spin-off from the *University of A Coruña*, was born in order to evolve and industrialize the original prototype of the server.

The initial goal of the project was to design and develop an innovative server to provide *VoD* services to the clients of the regional cable operator, at that time about forty-thousand subscribers living in several cities of *Galicia*, a region located in the north-west of *Spain*.

Before starting the *VoDKA* project, the most important available products were analyzed by the development group, and the conclusion was that most of them were expensive, closed, non-scalable and non-adaptable solutions. Therefore, the proposed goal for the project was to build an extremely scalable, fault tolerant, multi-protocol, adaptable (to the underlying network topology and to the end-user protocols) streaming server.

This part of the thesis is divided into three chapters. In the first one, the main concepts related to *VoD* systems are explained, the most important *VoD* requirements are described, and the state of the art both in academia and industry is presented. In the second one, the *VoDKA* architecture is discussed into detail from different points of view, explaining how the system was designed and implemented in order to meet the requirements. Finally, in the last chapter, the proposed architecture and its implementation are evaluated following different procedures, like performance analysis, requirements fulfilment, or analysis of the problems found during the system development.

The goal of this part is to describe the real, complex, industrial case study that motivated the theories and tools explained in the rest of the thesis. In order to understand better these theories and tools, the chapter explains some relevant information related to *VoD* servers and, in concrete, to the design, the development, and the deployment of *VoDKA*.

Chapter 4

VoD servers

Contents

Before explaining the *VoDKA* development, the learnings of the project and motivating the use of formal methods in order to improve it, in the present chapter we introduce the *Video-on-Demand* (*VoD*) world.

First, the most relevant concepts are explained, giving definitions that are going to be useful in the rest of the thesis. Then, the system requirements for a *VoD* server are discussed and described. After that, a description of the state of the art on *VoD* servers, both in industry and academia, are explained. Once the goals of the system are clear and some existing products have been presented, in the next chapter, the software and hardware architectures of *VoDKA* are described, explaining their evolution during the different stages of the server development.

4.1 *Video-on-Demand* definitions

After the massive consolidation during the 1980s of the traditional television as a basic element in the daily life of millions of people, the media industry has been continuously looking for more complex, flexible and interactive services. The cable and satellite technologies, developed in the 1970s, grew a lot during the last two decades of the 20th century, opening new options in the kind of services that could be offered to the end-users. Nowadays, most of the digital platforms offer different kinds of *Pay per View* services, in which the user can choose among several different pieces of content at one of the pre-fixed timetables they are shown.

The development is pointing clearly to more interactive services, where the user would have the opportunity to select any piece of content from a list of media pieces available in the server. The service could be requested at any moment in time, without any kind of pre-established scheduling. This is where *VoD* services appear into scene.

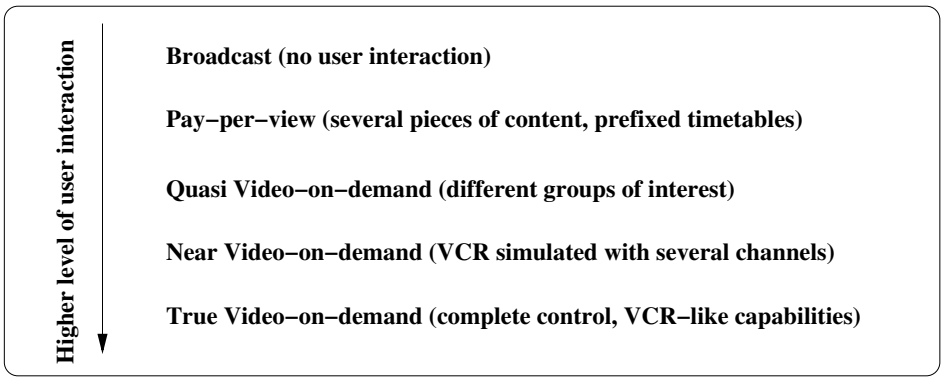


Figure 4.1: Levels of interaction according to Little and Venkatesh

This futuristic *VoD* concept has been categorized as the fifth level of services by Little and Venkatesh [LV94]. They propose a division of interactive digital services into five categories based on the level of user interaction, going from almost no interaction to the more advanced features: *Broadcast* and *Pay-per-view* are the ones already explained before; *Quasi* VoD refers to the services in which the users can perform simple temporal control activities by changing from one group of interest to the other; *Near* VoD services are those in which functions like fast forward or reverse are approximated, with normally several channels that are providing the same content within a small time difference; and *True* VoD services, in which the user has complete control over the session presentation, having VCR-like capabilities. In this thesis, *VoD* stands for the latter, most advanced, form of interaction.

In practice, VCR-like capabilities are very complex to implement, and they are normally simulated using multiple channels, so pure *True* VoD services are not very frequently found.

When we talk about *VoD* systems, services, applications and servers, we are using concepts that could be easily confused or interchanged. In Fig. 4.2, the global idea of a *VoD* architecture is presented, specifying the concrete names for each of the components that are going to be used in the rest of this thesis. The whole *VoD* system architecture is divided into three big components: the *VoD* server, which is in charge of storing the media content and performing the actual streaming; the *VoD* applications, that are built on top of the *VoD* services provided by the former component; and the management subsystem, in charge of dealing with the user accounts, authentication mechanisms, and that kind of services. In some *VoD* systems, this last component is part of the applications, but it is presented here for clarity reasons as a separate subsystem which can be accessed by the user applications.

Thus, on top of a *VoD* system, different multimedia applications can be implemented, such as *Movie-on-Demand*, distance learning, interactive shopping, interactive *News-on-Demand*, etc. It is important to distinguish between the *VoD* server technology, that is going to be discussed in this chapter, and the services offered to the end user, that are built as applications on top of the *VoD* server. The pieces of content available for the different applications in the *VoD* server are normally called *Media Objects*.

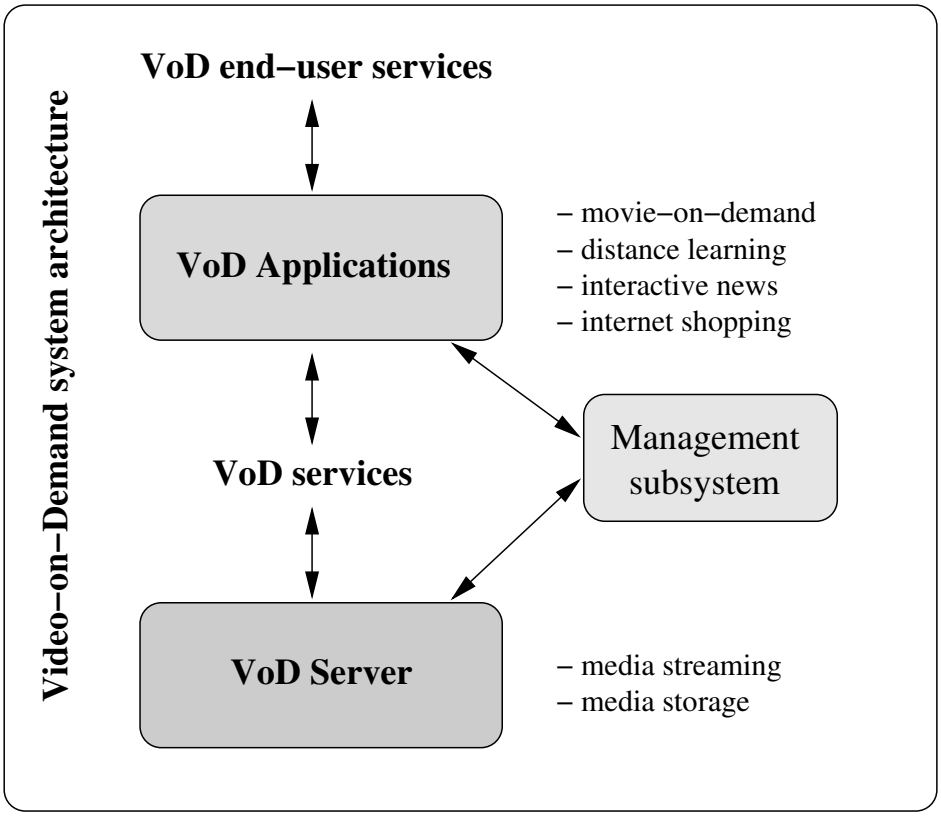


Figure 4.2: Main components of a generic *VoD* architecture

A (True) *VoD* system provides video services in which a user can ask for any *Media Object*, from the pieces available in the server, at any moment, having capabilities over the content similar to the ones provided by a normal video recorder. This means that all the pieces of content in the video server are, at least in theory, available at any time for being streamed to the end-user (via the selected application). Therefore, the request will be satisfied if there are enough internal resources available, without the need for any kind of temporal constraint and without pre-established scheduling.

This thesis is only considering the *VoD* server part of *VoDKA*, not the applications providing services to the end-user. Nevertheless, sometimes the analysis of those services is needed in order to learn more about how the server should behave, as it is done in the next chapter, when explaining the *VoD* server requirements.

4.2 *VoD* server requirements

Even though *VoD* servers (and the underlying technologies) can have very different nature and design level requirements, depending on the concrete needs of the application that is going to use them, they share a set of commonly agreed requirements that should be fulfilled in most of the systems of this kind. At least, all of them are commonly agreed as important features for a *VoD* server.

In this section, these requirements are listed and explained. In the remaining

chapters of this part, we refer to them when clarifying the choices made in the software architecture.

Before discussing the functional and performance requirements, it is important to think about the different kind of services the system should implement, which could be seen as the *use-cases* for the software we want to build. The simplest *use-case* would be a *Movie-on-Demand* application, in which the user selects a movie from the list of titles available. The list of movies should be big enough to compete with traditional video rental stores, and a popular movie could be requested by a large amount of users in a short period of time, normally with a very irregular distribution over the different hours of the day and over the days of the week. It can be acceptable to deny a user request for streaming a movie if the server remaining capacity is not enough for handling a new stream. On the other hand, as soon as the service starts, the user should receive all the requested content without undesired interruptions, which can mean several hours of reliable continuous streaming.

Other applications can have different usage profiles: in *News-on-Demand*, for example, the requested pieces of content are going to be shorter, and the response time needs also to be reduced compared with what could be acceptable in the case of *Movie-on-Demand* applications. For shorter fragments one demands shorter waiting time. There is a special case of *Movie-on-Demand*, when multi-camera movies are offered to the end-user, where each of the users connected are going to request several pieces of content at the same time.

When designing the architecture of a *VoD* server, it is important to keep in mind that, most of the time, improving a given requirement can result in a conflict with another. For example, an increase in the system storage capacity (needing therefore more storage resources) could raise too much the cost of the whole system, and at the same time may increase the average response time for a given user request, because there would be an increase in the number of *Media Objects* that would be stored in slower devices. Of course, as is always the case with software projects, all the system requirements are closely connected with each other, and a good design solution would be able to find a balance, giving a good average answer to all of them.

Normally, not all the *Media Objects* have the same profile of use, and this knowledge can be used by the system engineers to design the software and hardware architecture in a clever way. Also, as already explained, the system is not used everyday and at every moment with the same profile of use, and this knowledge can also be taken into account by the engineers to improve system behaviour and even to stimulate a change in the use (a simple idea would be to make offers in the times of the day where the system is less used, or to make special offers for those *Media Objects* that are *easier* to stream for the system).

Given the variety of the explained use situations, the main requirements for *VoD* servers are the following ones:

1. **Huge storage capacity**: *The server should be able to store thousands of* Media Objects*.*

   In most *VoD* applications, the server needs to provide the user with access to a large number of *Media Objects*, which must be stored in some way inside

the server. As an example, the size of a movie encoded with the current compression technologies, with a reasonable quality, can be easily close to one gigabyte (more if we want broadcast quality); a common small size movie rental store can have about a thousand titles available, which means that about one terabyte would be needed to store a similar amount of media in the digital server. Taking into account that re-encoding content is too slow for doing it on-demand, it is not uncommon to store the same content in different qualities (i.e., in different file sizes), having them available for the users of the system; this produces an increase in the storage capacity needed.

2. **Large amount of concurrent users**: *The* VoD *server should be able to attend requests from thousands of users at the same time, each of them possibly asking for several* Media Objects .

   Due to the nature of the *VoD* applications, a lot of users can potentially request several pieces of media at the same time, and the server should be able to accept a great amount of them and to provide them the piece of content they request. This should be done maintaining a reasonable quality of service for all the users. It is even possible to receive several requests concurrently from the same user, as it happens with the multi-camera movies, in a *Movie-on-Demand* service.

3. **High bandwidth**: *The* VoD *server should be able to stream thousands of high bandwidth pieces of content at the same time.*

   As we have already explained, a *VoD* server needs to be able to serve a large amount of concurrent users, and most of the *Media Objects* are stored in the server with a quality and format that require high bandwidth for streaming them. Therefore, it is easy to conclude that the system software and hardware architecture should be designed in a way that is able to provide a high total throughput.

4. **Low response time**: *The* VoD *server should have a low response time, which should be predictable at the time the user performs a request.*

   As an example: in *Movie-on-Demand*, a user requesting a movie could accept the situation where the system asks for waiting during several minutes before starting the actual streaming, but would probably not accept a waiting time exceeding a quarter of an hour. On the other hand, in a distance learning class, we could imagine that more than half a minute (or even less) of waiting time would be too much for an average user. For other kind of services, like interactive *News-on-Demand*, response time should probably be at most a few seconds for the *Media Objects* accessed by the users. The response time in this kind of *VoD* servers, therefore, needs to be reduced as much as possible, this being specially critical for some of the applications and services provided. At least, the system should be able to create, using different kinds of statistical estimations and architecture status analysis, a prediction of the waiting time for each user requesting any *Media Object* to the system.

5. **Availability and reliability**: *The* VoD *system should be both highly available, being able to keep working in the presence of errors, and reliable, mini-*

mizing the presence of software and hardware errors.

Clements, Kazman and Klein [CKK02], define availability as *the proportion of time the system is up and running. It is measured by the length of time between failures as well as how quickly the system is able to resume operation in the event of failure*, and reliability as *the ability of the system to keep operating over time. It is usually measured using the time to failure.* Both are key requirements for *VoD* servers.

The average users of most of the services and applications that run on top of a *VoD* server expect 24x7 uptime. This is partly caused by the influence of the traditional television services, which are normally working also permanently almost without user detectable failures. Users expect streaming interruptions due to technical problems to be very uncommon.

The normal service times for this kind of applications vary from minutes to several hours of continuous streaming, a long time during which the user should not notice any reduction in the quality of the stream, due to any kind of hardware or software failure. A degraded mode, even though, could be acceptable while the system is recovering from a failure, depending on the concrete application (quality critical systems, like *VoD* in a hospital [MGP+01], could not work with low quality, being a better alternative to stop the service in that case and notify the responsible).

It is important to distinguish between *user service availability* and *system availability.* The former is related to the resources available at the moment the user requests the streaming, and can be seen as a different definition for the combination of the *large amount of concurrent users* and the *high bandwidth* requirements. The latter is the requirement explained here, related to the ability to keep running and answering user requests (even saying that the requested *Media Object* cannot be sent because all the resources are used); and with the ability to maintain the existent connections with the user, even after some software or hardware failure.

Therefore, the system should have a small amount of software errors (number of bugs reduced, reliable system); should be able to recover from them, and from hardware errors (hard and soft fault tolerance); and should be possible to add new functionalities or modify the existing ones without stopping the system (hot code and hardware swap).

6. **Upgradability and maintainability**: *Adding new software or hardware features, or correcting or changing the already existent ones, should be done in a simple way, without needing to stop the* VoD *server.*

   Multimedia technologies, specially the big amount of protocols and formats involved, have been constantly evolving during the last years. This means that frequent upgrades adding support for new features, formats or protocols to the server are carried out. Therefore, having an upgradable system is a very important feature nowadays.

   Besides, it is well known in software engineering that the biggest part of the cost of a software product is spent in maintaining the system running. That means that in a system easy to maintain, the overall cost is reduced. That

difference could be just saved, or spent in improving the software quality, depending on the needs of each company or project.

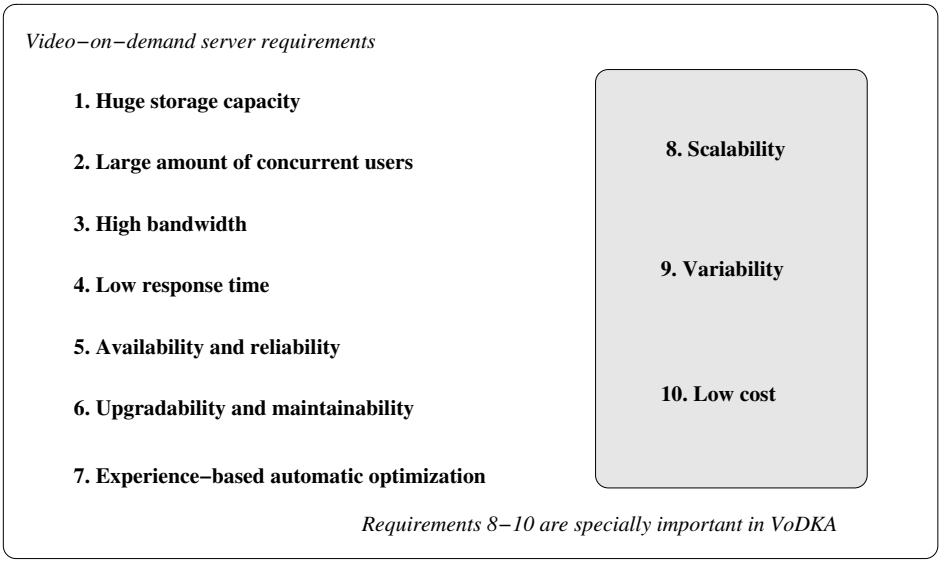


Figure 4.3: Summary of the main *VoD* server requirements

7. **Experience-based automatic optimization**: *The* VoD *server should be able to collect information from the users behaviour, and take them as an input for improving the internal scheduling algorithms.*

   After having a *VoD* server deployed for a while in a given environment, a lot of information about the user profiles and behaviour can be collected. Which *Media Objects* and with which timetables are more popular, how many users on average are expected for each day of the week, how long do users watch the streamed piece of content, etc. Also, knowing what applications the *Media Objects* are used by, gives interesting information about concrete requirements for each of the pieces of content stored in the server.

   This information can be used to influence the internal algorithms in charge of optimizing the server configuration for selecting the position inside the server in which the *Media Objects* are placed, the number of copies available, and so on. This feedback input can be provided by external management software, or collected by the server itself.

8. **Scalability**: *Adding new resources to the* VoD *server in order to improve the performance of the system, and removing resources when not needed, should be possible.*

   The software and hardware architecture of the system should be generic enough for being both downward and upward scalable, adapting it to the needs of every particular deployment. Scalability means that an increase in the performance requirements, such as the amount of concurrent users, the bandwidth, and the storage capacity could still be satisfied by adding resources to the system.

Downward scalability means that the *VoD* system should be able to work optimally in simple environments, where only a few users are going to request a small amount of *Media Objects*. In this kind of deployments, the storage capacity does not need to be so large, and the number of concurrent users is very limited (in fact the first three requirements of the list are not so important in those examples). The complexity of the server architecture configuration should be simplified in these cases, providing a system easier and cheaper to deploy and maintain. *The ability to support the production of a subset of the system*, is defined by Clements, Kazman and Klein [CKK02] as subsetability, and can be considered as a special case of downward scalability.

On the other hand, the same *VoD* server should be upward scalable for working in very complex environments, with a growing amount of concurrent users and storage needs. Adding more instances of the same software and hardware components to the deployed system should be possible at any moment, improving the overall system performance. This way, if a given company implements the streaming solution with the *VoD* server, but the capacity requirements increase at some point in the future, they do not need to invest in a new system, because adding more resources (new machines with more software components running on them) would satisfy those requirements.

9. **Variability**: *The* VoD *server should be adaptable to the end-user protocols and to the underlying network topology.*

   Variability is defined by Clements, Kazman and Klein [CKK02] as *how well the architecture can be expanded or modified to produce new architectures that differ in specific, re-planned ways*, and it is explained that this variations can refer to run-time modifications, compile-time changes, build-time reconfigurations, or code-time redesigns.

   Modern *VoD* servers should be designed for being deployed in modern networks, which are characterized by having different sections in the network topology, each of them offering vastly different features (bandwidth, response time, and other features). There is, actually, a strong need to optimize network resource usage, avoiding the bottlenecks in the slowest parts of the topology.

   System designers do not know on beforehand how the underlying topology is going to look like, and the system can potentially be deployed in completely different networks. There is also a need of flexibility in the redesign of the system deployment: the architecture should be generic enough for allowing this kind of redesign.

   Therefore, a *VoD* system should be variable, in the sense that it should be possible to adapt it, normally at deployment time, to the underlying network topology, making an efficient use of the available network resources.

   As already explained, the kind of protocols used by the end-user applications can vary a lot, and the server should be able to adapt its configuration, if needed, to support the new protocols demanded.

10. **Low (affordable) cost**: *The* VoD *system should have as low as possible development, deployment and maintenance costs.*

It could be stated that this is a requirement for every system, and that every system architect should have in mind the low overall cost as a goal, but it is discussed here as a special requirement because it was defined as a must during the first stages of the *VoDKA* development. One of the main goals was to construct a viable, competitive alternative to the very expensive solutions already existent at the moment the project was started.

When talking about cost, we are referring to reduce the costs in the different fields in which a system can be costly: reduction of the hardware cost; reduction of software development and maintenance costs; and reduction the network usage and requirements, being optimal in the use of bandwidth and connections required both for the internal and external communications of the server.

All the requirements in the list conditioned the design of the *VoDKA* server, but the last three ones (scalability, variability and affordable cost) where considered specially important and had a lot of influence in the software and hardware architecture and technologies selected for developing the system. More details on this topic will be found in Chapt. 5.

4.3 State of the art in *VoD* systems

In the latest years, the leading companies in the digital media sector have developed *VoD* related solutions. At the beginning of the project, the analysis of the available products gave the conclusion that most of them were expensive, closed, non-scalable and non-adaptable solutions. On the other hand, they are normally turnkey solutions, ready to plug and easy to run on a deployment, which can be seen as an advantage for some environments. Some of the main enterprise solutions and also the main initiatives in academy are discussed below.

There are several important criteria for analyzing the available solutions in order to compare them with the one presented in this thesis. Relevant differences can be found in the kind of operating systems that are supported by each solution, and the programming language the system is implemented in. Both features can have a big impact in the upgradability, availability or the scalability of the system. Another key factor is whether the server supports and uses extensively open protocols, which make the interoperability much easier and improve in general the quality of the system; open standards allow the implementation of ad hoc client solutions, without needing to acquire and use the proprietary ones from the same company that builds the server. The license of the source code can also have an enormous influence in the system flexibility and maintainability; *Open Source* solutions allow the optimization even changing and profiling the source code when necessary. The supported hardware is strongly related to the amount of bandwidth, the number of concurrent users, and other performance requirements. Some of the servers are designed with scalability and redundancy (for providing fault tolerance) in mind, but others are more targeted to a concrete deployment, and are therefore less flexible. The cost can vary a lot in the different options available, thinking in the TCO, not only in the deployment or acquisition costs. Some of the systems are targeted to a concrete database, and others are more flexible; the database

management system used can have a strong influence also in several important requirements.

Next, the most important features for the main solutions available at the beginning of the project are discussed. A more detailed description of them can be found in [SGVM00].

4.3.1 Enterprise Solutions

Some solutions were well suited for low bandwidth networks. The most popular ones were:

- RealNetworks RealVideo Server: it is based on the open standard RTP/RTSP and supports several operating systems and hardware platforms, and its code is not available as *Open Source*. It is very popular and widely used through the Internet.

- Microsoft Windows Media Server: mostly uses proprietary protocols, supports Windows operating systems, and their target hardware platforms.

Other solutions are more focused to LAN or MAN, with high bandwidth availability. The most representative ones are:

- Apple Darwin Streaming Server [App] It is the *Open Source* version of the Quicktime Streaming Server, and is in fact an efficient but not distributed or clusterable RTP and RTSP streaming server, without including distributed storage management.

- Oracle Video Server (OVS) [Ora98a, Ora98b], is probably the most widely used solution, with a client/server architecture, but with scalability limitations. It is available for Solaris and Windows NT platforms. It can provide real-time video feed and normal *VoD* service. It has a modular, client/server architecture. It supports file striping to improve the response time and RAID disk configuration to increase the security by using redundancy. It works with several video formats, both standard and proprietary ones.

- IBM's DB2 Digital Library Video Charger [WDRW99], enhances IBM DB2 Digital Library by delivering digital audio and video over the Internet or intranet. It is a solution very similar to OVS; it offers industry standard-based features, specially IP multicast standards (it uses RTP and it is protocol-ready for Real-time Streaming Protocol and Reservation Protocol). It supports the same open video formats as OVS. It also supports a high-capacity digital tape archive system for off-line storage.

- Kasenna MediaBase, an evolution of the SGI's WebForce MediaBase, which has a modular design, separation of acquisition, distribution and streaming functions, based on UNIX concepts.

- Philips WebCine Server [Phi], an MPEG4 streaming server based on *GNU/Linux*.

- Cisco IP/TV [CS00], closed solution with some high level tools oriented to e-learning.
- Sun's system: StorEdge Media Central [Sun].

There were also black box solutions, which consist in a combination of specialized hardware with some of the described products.

In the latest 5 years, three companies have being key factors in the market of the *VoD* servers: Concurrent (MediaHawk On Demand Platform), SeaChange (SeaChange On Demand Platform) and nCube (nCube On Demand Platform). All of them offer commercial products that have been massively deployed all over the world and offer a flexible platform. However, they still represent closed and not always affordable solutions that still represent a different approach to the one selected for the development of *VoDKA*.

4.3.2 Academic World Solutions

In the academic environment, most of the projects studied at the beginning of the *VoDKA* development offered experimental solutions. Some of the most representative ones, that ware taken into account for the *VoDKA* design, are:

- The Stony Brook Video Server Project [CVV97a, CVV97b, MV96] is oriented towards building a distributed and scalable video server application that provides indexing, searching and video streaming to clients over the network, it was at Stony Brook, New York, by Andrew V. Shuvalov. Possible video sources were VCR, Satellite TV and cable TV, and special attention has been put on data integrity and fault tolerance.
- In [DHLV96] an alternative solution, using a shared memory architecture and some aspects about *VoD* server design and a server model based on a modular software architecture are studied. Different methods for disk striping are also compared at the storage layer to balance advantages and drawbacks for each technique. The system is divided into three main modules: storage subsystem, processing and routing subsystem, and network subsystem. For instance, the storage subsystem is presented as a hierarchical structure based on a great number of disk units, CD-ROM jukeboxes, and magnetic or optic units. The system is based on a *VoD* server developed at the University of Minnesota.
- One important inspiration for *VoDKA* was described in [CT97a], where a hierarchical solution for building multimedia servers is analyzed. The proposed architecture was based on three fixed levels for streaming, caching content and storing a big amount of *Media Objects*.
- Other research initiatives that were studied include: in [BGM95, AO92, Ber94, CKY94] some research on the disk subsystem at the *VoD* servers are carried out. In [AO92, GC92] topics related to real-time video playing, or multiple audio channels, have been studied. In [RV92, RV93], authors present methods to store audio and video in these systems and how to introduce storage patterns that can reduce disk usage. In [LS93] a disk-storage

based system which can serve multimedia requests is presented. In [Ber94], a striping method with an efficient mode for sending video and audio objects with different bandwidth requirements is presented. In [LL96] the scheduling and file replacement policies of a hierarchical storage system are discussed. In the paper, the architecture is composed by a tape store, a mechanic arm, a secondary level of disks, and a primary storage at memory. In [CT97b] an theoretical study of different concepts related to the creation of a hierarchical server is presented. The demand, video files, performance requirements of each application and type of interaction characteristics are considered. Three main performance objectives are stated: to carry out the performance requirements, to get effectiveness at minimum cost, and the robustness and scalability.

More recent research results on the field of the *VoD* servers, in some sense related or influential to the contents of this thesis include: in [LC03], an scalable and efficient *VoD* platform is presented; GloVE [PAI02] (Global Video Environment) is a distributed environment for low cost, scalable *VoD* systems that uses a cooperative video cache shared by several clients; in [STH06] the design and implementation of a video broadcasting server based on a technique called Striping Broadcast are described; P2VoD [DHT04], a proposal for a fault tolerant *VoD* streaming in Peer-to-Peer networks; in [HF04], a proposal for using dynamic load balancing among *VoD* servers for reducing the average response time of requests is proposed; in [FBA03] three fault tolerant models for *VoD* services, trying to guarantee continuous streaming even after server failures, are discussed; and in [THS03], the focus is on using caching for streaming over the Internet using an overlay architecture consisting of caching servers.

Chapter 5

VoDKA architecture

Contents

In order to satisfy the system requirements, an innovative solution, based on a flexible, distributed and hierarchical storage system [BGF+01, GAS03b, GBF05], developed in *Erlang/OTP*, and built on top of *GNU/Linux* clusters composed by commodity hardware [BG99], was proposed for the *VoDKA* server.

In this chapter, the *VoDKA* architecture is explained following *The 4+1 View Model of Architecture* defined by Philippe B. Kruchten in a very influential paper [Kru95] published in 1995. The *4+1 Model* explains the system architecture from five concurrent complementary views, each of them explaining a different set of issues. The *logical view* explains how the system is decomposed into a set of abstractions, concepts and components, and how they relate to each other; the *process view* concentrates in which thread of control each of the operations in the system executes, therefore addressing concurrency and distribution; the *development view* shows the organization of the software source code and documentation, and focuses in the development process and methodologies; and the *physical view* details how the software is deployed in a network of computers, relating therefore software to the actual hardware where it runs. These four views are derived from

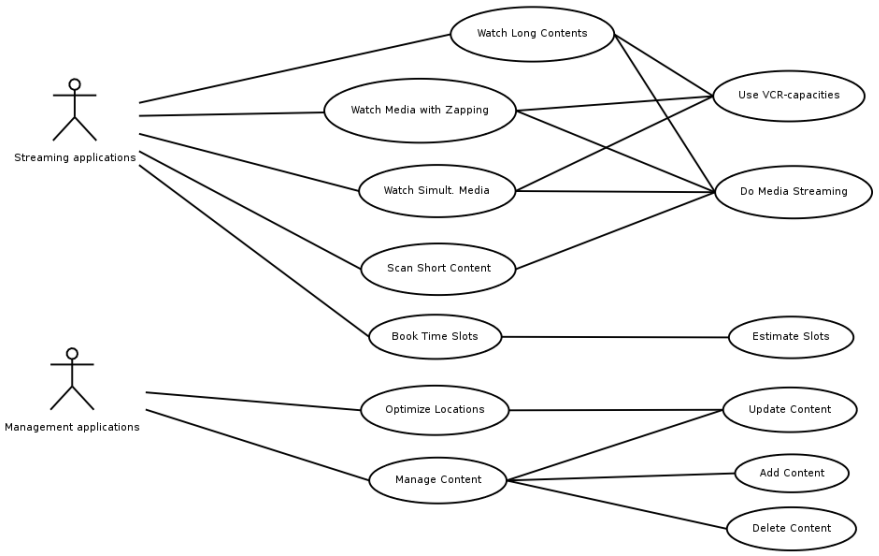


Figure 5.1: Main *VoDKA use-cases*

a selected group of very representative *system* use-cases, which conform the *+1* view. Although the model was originally designed for systems developed with the object oriented philosophy, we use it adapting the different views in order to make them adequate for the process-based distributed functional system we present in this thesis. Besides, we have added a section, containing the *development view*, information about how the system was developed, including tools and details about the underlying architecture.

5.1 System *use-cases*

The *use-cases* of a system describe its main functional requirements, explaining how it ushould interact with the end-user. In the case of a *VoD* server, the end-user is not interacting directly, but through one of the software applications that are built on top of the server. From a high level perspective, each of the *VoD* applications define a set of closely related *use-cases*.

In most of the applications built on top of a *VoD* server, the *behaviour* of the end-user follows a similar procedure: the user logs into the application and authenticates into the system. The authentication method can vary from simple login/password to some more sophisticated technology like the ones using certificates. Then, the application shows a list of pieces of content available (after querying the *VoD* server to check if they are really available). After receiving the list, the user selects one or more pieces of multimedia content and interacts with the system during a time that can vary from minutes to several hours. Finally, the session is closed and all the resources are freed. The authentication, session management and related parts of this scenario are handled by the application part, without any kind of interaction from the core *VoD* server. In real scenarios more complex technologies like virtual tickets or DRM (Digital Right Management) need to be taken

into account, but we are not going to go into those details here because they are not relevant for the rest of the thesis.

From now on, we concentrate in the *use-cases* for the *VoD* server itself. The most important ones are shown as a UML diagram in Fig. 5.1. The first level of relations associates the two main roles (not played directly by humans but by two kinds of high level applications) with *use-cases*. The second level of relations represents standard *uses* associations between *use-cases*.

- **Long *Media Objects* visualization**: the user gets into the server and, after browsing the available contents, chooses one *Media Object* (a movie, sports game, documentary, etc.) and visualizes it until its end. The user would accept a bigger response time (system latency) than in other *use-cases*, but the system should be able to at least provide an statistical estimation of the waiting time; unknown and long waiting time would not be accepted. As the *Media Objects* are visualized without interruptions, for up to several hours, the system needs all its fault tolerance features in order to keep the quality of service for such a long period. Depending on the case, VCR-like interaction would be used (e.g., sports game visualization with repetition of the main parts and fast forward during the boring minutes) or not (e.g., a regular high-quality movie watched without any kind of interruption).

- **Media zapping**: the user gets into the server and asks sequentially for several (possibly long) *Media Objects*, watching them for a while, trying to get some basic information about them, and then going to the next piece of content. Short latency is required, and a lot of fast resource management is involved inside the server. The users could be interested in finding some concrete part of the *Media Objects*: searching facilities would make this much easier and faster, and even less resource-intensive for the system. The streaming time is not normally very long for each piece of content, but quite a lot of VCR-like interaction could be needed.

- **Several movies at the same time**: the user asks at the same time for several *Media Objects*, that are going to be streamed in a synchronized way. Whenever any kind of VCR-like operation is performed, it affects all the streams. This involves resource booking for all the *Media Objects* as a unique stream, because the user would not normally be interested in watching only some of them. This *use-case* is specially frequent in multi-camera movies visualization or multi-camera sports games retransmission (the user selects which cameras wants to see among a list of available *points of view*).

- **Short content scanning**: in this *use-case*, the server is required to stream a large amount of very short pieces of content, requested by the user sequentially. Short latency is a very important feature, as it is fast resource booking and freeing (the length of the *Media Objects* is known, as opposed to the reproduction time of the cases where the media is not completely streamed). Almost no user interaction is expected during the streaming of the *Media Objects*. This *use-case* takes place, for example, when *News-on-Demand* applications are using the *VoD* server.

- **Content management**: the administrators of the system need to keep the content of the *VoD* server up to date. Old content could be removed, and new pieces of content are constantly being added to the system. This involves the update of all the internal metadata used by the scheduling algorithms. The administrators would access this functionality of the core *VoDKA* system through high level management applications.

- **Optimization of content location**: for several high level reasons (even sometimes commercial ones), the administrators of the *VoD* server could be interested in having very good performance for a concrete *Media Object* or a group of (possibly related) *Media Objects*. In order to do that, they would connect to the *VoD* server and describe the kind of content they want to improve the performance of, allowing the *VoD* server to change the location of those *Media Objects* or booking a special set of resources for them. The performance of the server for a concrete set of *Media Objects* could be improved for a week, only during a day, forever, or even depending on the time of the day (e.g. the *Media Objects* targeted to children should have a very good performance in the afternoon, but they are not normally going to be requested at night).

- **Slot booking**: it is not always the case that the users want to receive immediately the media stream they are interested in. Sometimes, they prefer to search the contents, select a given *Media Object* (or a group of them, as explained in the previous *use-cases*), and give to the system an approximate starting time. The *VoD* server should ensure that the needed resources are going to be booked in advance, so that the streaming could start at the time selected by the user.

The *VoD* server, therefore, has to be able to perform the following main tasks (they can be seen as services for the applications), as part of the described *use-cases*:

- Calculate the list of the available content matching some conditions, offering search facilities.

- Calculate the predicted response time for a given piece of content.

- Stream a given piece of content to the end-user after allocating the required resources.

- Provide search and VCR-like capabilities on the contents that are being streamed.

- Add, remove and update content.

- Optimize resources for a set of *Media Objects*.

- Book a concrete time slot (ensuring that resources will be available) for streaming a given piece of content to the end-user.

These functional requirements, together with the ten generic non functional requirements for *VoD* servers described in Chapter 4, were the goals taken into account for designing the software and hardware architecture that is going to be described hereafter.

5.2 General design ideas

Given the requirements, the *VoDKA* system was conceived from the beginning as a flexible architecture of specialized components. The existence of specialized components gives answer to the different kinds of requirements, sometimes even contradictory ones, we have already explained.

For example, if the system needs to have a huge storage capacity, and at the same time should be able to reduce the response time and saturate a high bandwidth, solving this without specialized layers or components (monolithic or homogeneous distributed architectures) would mean to use very expensive hardware (contradicting the cost requirement) and to complicate the software pieces in a way that they would be very difficult to evolve and maintain (contradicting again the proposed requirements).

The solution was to create components specialized in satisfying, each of them, some subset of (non contradicting) requirements. This is the case of components like a very fast streaming layer capable of offering a very good performance handling a lot of concurrent connections; and a cheap massive storage component able to store a lot of *Media Objects*. The components cooperate in an architecture based on the concept of delegation, where a flexible cache-based hierarchical configuration keeps the more popular *Media Objects* as close as possible to the fastest layers.

The communication inside and among the components is based on a message API kept as simple and homogeneous as possible. Almost the same messages are sent through the architecture from the user to the layer where the *Media Object* is found. In their way trough the architecture, the messages are filtered depending on the resource availability and local scheduling decisions, normally based on heuristics and a concept of *cost* that keeps the system load as balanced as possible.

The fact of using very flexible components, internally based on design patterns that have a direct relation with *Erlang* implementation patterns (called *behaviours*), is one of the keys of the architectural flexibility. Some of these design patterns implement part of the internal scheduling algorithms of the *VoDKA* server. As we will explain in Part III of this thesis, the server has a global internal scheduler, spread all over the system processes and components. Knowing the design is necessary in order to understand the approach we have taken to apply formal methods for improving the quality of the software.

5.3 Logical View

The logical view of our system describes the main components of the *VoDKA* server and how they collaborate. First, some implementation patterns that are going to be used recurrently all over the system architecture are presented. Then, the main components are explained, defining their API, and how they are implemented.

The philosophy used is not object oriented but high level, distributed, process oriented, message passing. This is coherent with the concrete programming platform the system was developed with (*Erlang/OTP*), but all the explanations in these sections, being for a concrete philosophy, are independent from the actual programming language the system is implemented in.

The *VoDKA* server was designed and implemented following a distributed functional programming philosophy. The server consists of a set of distributed software components which communicate by message passing using a well defined API. Inside each of those components, a set of concurrent specialized processes collaborate also by exchanging synchronous or asynchronous messages in order to perform the internal tasks. Inside the processes, a functional approach, without any side effects apart from the inter-process communication, is used. Some of the processes inside the components are very similar, and carry out the same kind of tasks with some small differences in their internal *behaviour* or the process API. In order to abstract the common *behaviour*, design patterns are defined and used during the design and development of the server. Some of them are generic patterns that are sometimes provided by the development platforms. Some others are more specific *behaviours*, only valid for a concrete system or a family of similar systems. We will use the terms *pattern* and *behaviour* as synonyms in this context.

5.3.1 Distributed design patterns

The following design patterns, some of them already available in *Erlang* and others developed *ad hoc*, are widely used all over the *VoDKA* system:

1. Supervisor-worker: this pattern allows the processes implementing it to supervise other processes, receiving any kind of failure information and being able to react on that, following the provided policy. They allow the creation of supervision trees, which are a way of structuring the processes of a system in a tree, based on the relation between workers and supervisors. Workers perform the actual computation, implementing the actual logic of the system. Supervisors monitor the workers and implement a set of policies (provided for each case as a plug-in for the pattern) for deciding what to do when something unexpected happens to them (e.g. a process crashes because of a software or hardware failure). As it is possible for a supervisor process to act as a worker for another supervisor, the final structure is a tree with the workers on the leaves. This structure allows the creation of supervisors for subsystems (a process that defines the supervision policy for a set of related processes which are part of a subsystem). The pattern is, therefore, very flexible and provides a way of introducing the fault tolerance mechanisms of a distributed system in a homogeneous manner. A supervisor node has a list of child processes, and is in charge of starting, stopping and monitoring them, keeping them alive by performing the actions defined by the policy chosen for each case. The supervisor pattern and the creation of supervision trees provide a way of having fault tolerance inside the system processes and components. This pattern can be seen as an specialization of the master-slave pattern already defined by Buschmann [BMR+96]. Fig. 5.2 shows an

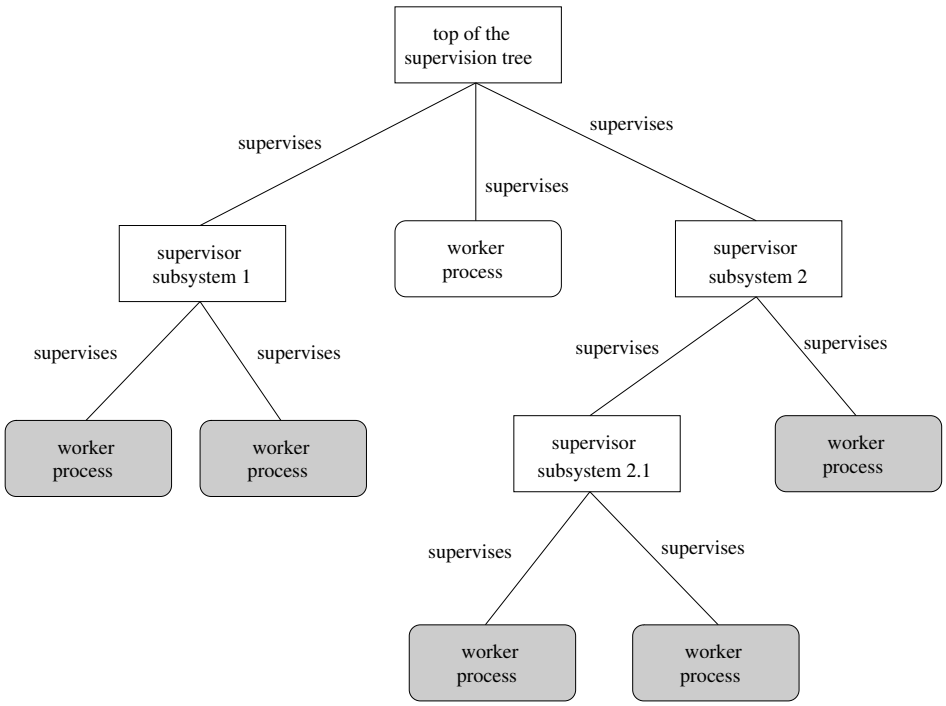


Figure 5.2: Supervisor-worker pattern: example of a supervision tree

example of a supervision tree. The use of this pattern is directly related to the fault tolerance requirement explained in the previous chapter.

2. Event handler and event manager: in distributed control systems, it is often interesting to have an event manager (or a group of them specialized in different kind of events) per each group of processes running in the same machine (a *node*). Any kind of event (including alarms, error messages, or other information that should be logged or processed centrally) would be sent to the event manager, who would resend it to all the event handlers associated with it, which know what to do depending on the kind and content of the received event. Event handlers can be attached to several managers, and can be in different nodes. They can also be subscribed to only a set of message types. The central event manager can be seen as an abstract pattern, with standard mechanisms for starting it, sending notifications and attaching concrete event handlers, which would have a state and a set of algorithms for evaluating each of the messages received. The configuration of the pattern is shown in Fig. 5.3. This pattern is related to the idea of making the system easier to maintain, introduced as part of the requirements for *VoDKA*. However, the solution used inside *VoDKA* is similar but not exactly that one. The Monitor pattern uses the observer design pattern to allow the creation of a process similar to the one described as event manager. The messages received are classified in groups, and sent to all the processes that had been previously subscribed to any message of that class, using the observer pattern interface for that. In Fig. 5.4, we an see an scenario where an observer subscribes to a monitor in

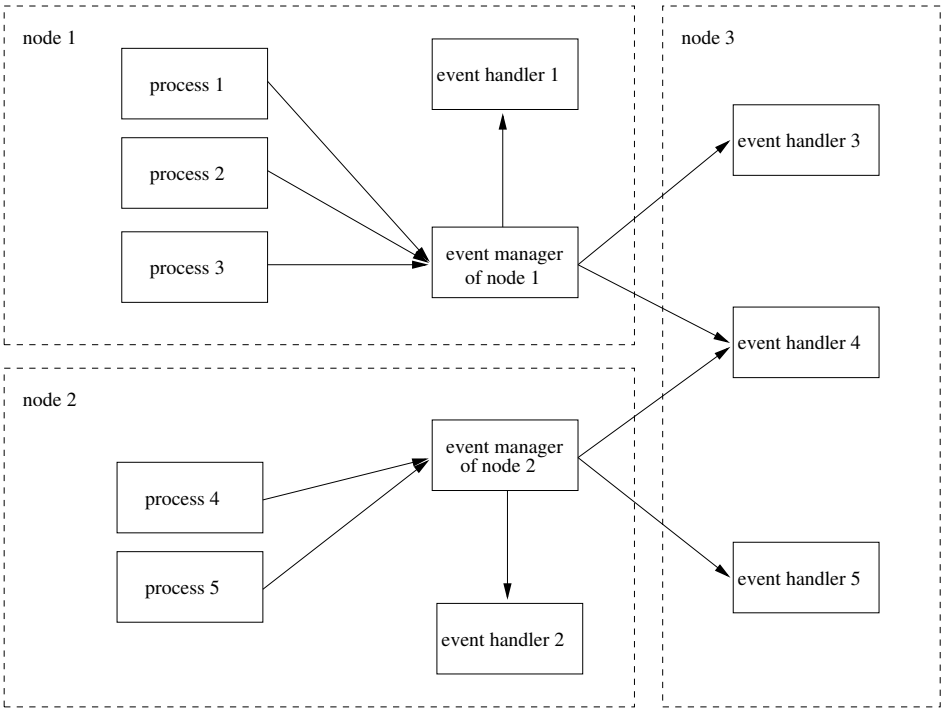


Figure 5.3: The event handler and event manager pattern

order to receive notifications from a given process type. More information on the use of this pattern inside *VoDKA* can be found in [VGS+01b].

3. Generic server: in most message passing based distributed systems, some variation of the client/server model is used. All over the concurrent process based architecture, several processes act as clients of another one. They send messages, using the server predefined API, and wait for the answer (synchronous operation) or continue their normal *behaviour* (asynchronous operation). On the other hand, the server process, after receiving the message with the request, performs some internal computations (functional evaluations), changes its local state, and sends the answer message back to the client, if required. The generic framework is always the same, and the differences from one server implementation to the others is the way the new process state and the answer message are computed. Most of the server *behaviour* can be then abstracted and defined as a generic pattern, to be reused all over the system architecture in a easier and homogeneous way. The schema is shown in Fig. 5.5. This *behaviour* provides a way to store state information in each of the processes of a system without global memory, and is heavily used in distributed programming. A developer using this pattern only needs to describe what happens when a message of each of the possible kinds is received, i.e., what is answered back to the client and how the internal state is updated. Using generic servers some code is reused, and the new code is simpler. Reusing code that has been working already in previous projects and checked by different developers increases the quality of the system and therefore reduces

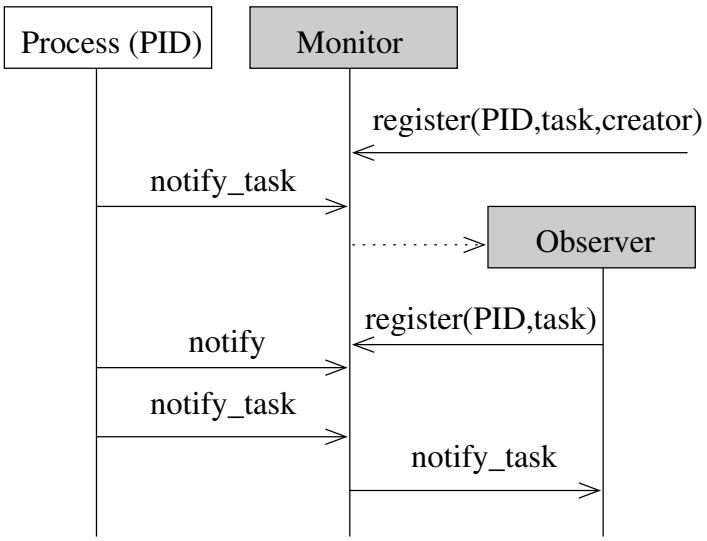


Figure 5.4: Example of the monitor pattern used inside *VoDKA*

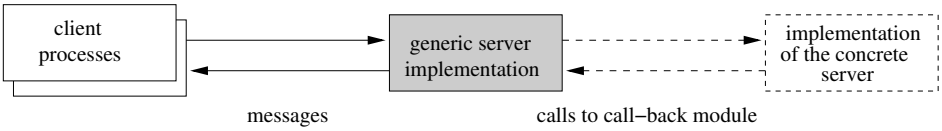


Figure 5.5: The generic server process pattern used in *VoDKA*

the number of errors.

4. Generic FSM: sometimes, the *behaviour* of a given process is very similar to the generic server described above, but the actions performed (both the new state computation and the answer message) depend on the concrete state of the server process. Therefore, for each state of the process, a potentially different message can be sent as answer and a different new state is computed, even for the same input. This can be seen as a special case for the generic server where even more *behaviour* can be abstracted and generalized. The user of the generic finite state machine has to parameterize the abstraction with the concrete states and the functions for calculating the answer message and the new state, in a similar way as it is implemented for the previous example pattern. The described pattern provides a clean way to describe finite state machines avoiding ugly and complex source code, making cheaper and easier the later maintenance and the addition of new features to the system.

5. Reflective Generic Server (RGS): is an extension of the generic server pattern with metainformation and a standard API that lets any other process ask to a RGS about its capabilities and its internal state. One interesting use of this pattern is the creation of components that can be asked for information about the services they offer. In *VoDKA*, for example, a service invocation interface has been developed using these introspection capabilities. This is also very

useful for system monitoring. The process metainformation is introduced by the developers (both in the design and source code), and can then be explored by other processes in the system. A process can at any moment ask a reflective server for the list of messages it understands, the parameters they carry, and other related metainformation. The same idea could be used for other kinds of patterns, creating that way Reflective Generic FSMs, or Reflective Event Managers, for example. The RGS is inspired in the idea of introspection, present nowadays in most of the object-oriented languages, which refers to the capacity of the language or platform for offering information about the public methods of an object and their parameters.

6. Trader: The Product Trader is a well known design pattern [BR98a]. In the *VoDKA* server, the internal architecture of the system, as it is going to be explained later, consists of very flexible components which can be plugged in different ways (software configurations). Therefore, the internal protocol messages need to be understood by all the components (which means it should be understood by the processes implementing the API for the component, described in Sect. 5.3.2). Instead of repeating all the structure, the protocol implementation is abstracted in a design and implementation pattern that trades in order to obtain the source and destination for each of the transmissions inside the system. The pattern also abstracts the idea of the chain of responsibility pattern that connects the components of the system together, and also the processes inside a component. This chain of responsibility establishes that for any request, the process or component is going to try to handle and answer it locally, and in case this is not possible, the pattern knows how to forward the request and wait until it has been solved by the following levels of the chain. This pattern is used in *VoDKA* as a particular extension of the RGS. Summarizing, a Trader allows the easy implementation of processes that receive requests specified by a standard message API, and try to satisfy the request or, if that is not possible, know how to delegate it to a different process.

7. Scheduler: when a *Media Object* needs to be streamed, it is often the case than several processes can be used as the source node for the streaming.In order to decide which of them is going to be chosen, the *cost* of each of them is calculated when the messages are sent through the system. The cost gives information about how good or bad for the global load of the server is to select each of the possible candidates. For calculating the cost, the intermediate processes need to be aware of how it is propagated and updated. In the scheduling processes of the system, two functions are evaluated each time one of that messages are handled, one for updating the cost, depending on the available resources of the component, and the other one for selecting which possible are going to be filtered out. Thus, in all these processes, abstracted to the scheduler *behaviour*, the functions *filter* and *update_cost* are implemented. A new design pattern, called scheduler, an extension of the previously mentioned `trader`, is used in *VoDKA* for this purpose. Fig. 5.6 shows an example of a trader process that filters and updates the cost of the answers received from other two processes it is connected to.

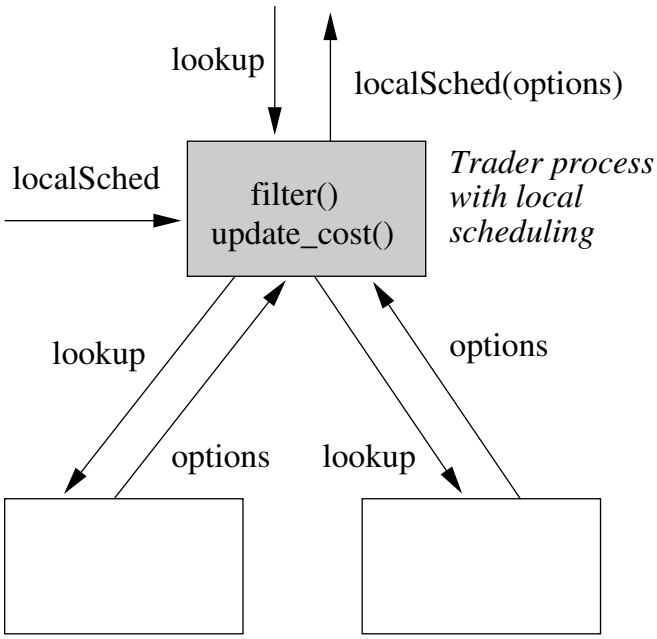


Figure 5.6: Example of a *scheduler* process in *VoDKA*

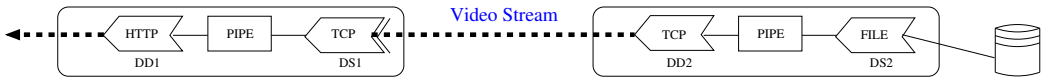


Figure 5.7: Pipe & Transfer patterns used in *VoDKA*

8. Pipe & Transfer: they are a data movement abstraction for the internal data communication [GAS03b]. A *pipe* is a process that has as its creation parameters the source and destination protocols (implemented as call-back modules), and some general options about the transmission. The source and destination modules need to implement three mandatory functions: `init` (protocol initialization), `proc` (read and write) and `done` (destructor). A transfer is the connection of two pipes, where the destination of the first pipe is connected using some transmission protocol to the source of the second pipe. By using the *transfer* abstraction, any two components of the server can be easily interconnected in order to stream *Media Objects*. Indeed, using these patterns, a whole *VoDKA* server can easily be configured in order to play the role of a source for the storage layer of other instance of *VoDKA* (providing a very flexible way of connecting several servers for collaboration). Fig. 5.7 shows a transfer composed by two pipes; each pipe has a source and a destination call-back modules.

9. Resource constraint: as it is going to be explained in the physical view, the *VoDKA* hardware architecture configurations can be quite heterogeneous. One can, for example, run each of the components in a different set of machines; in this kind of systems, a lot of access and communication constraints

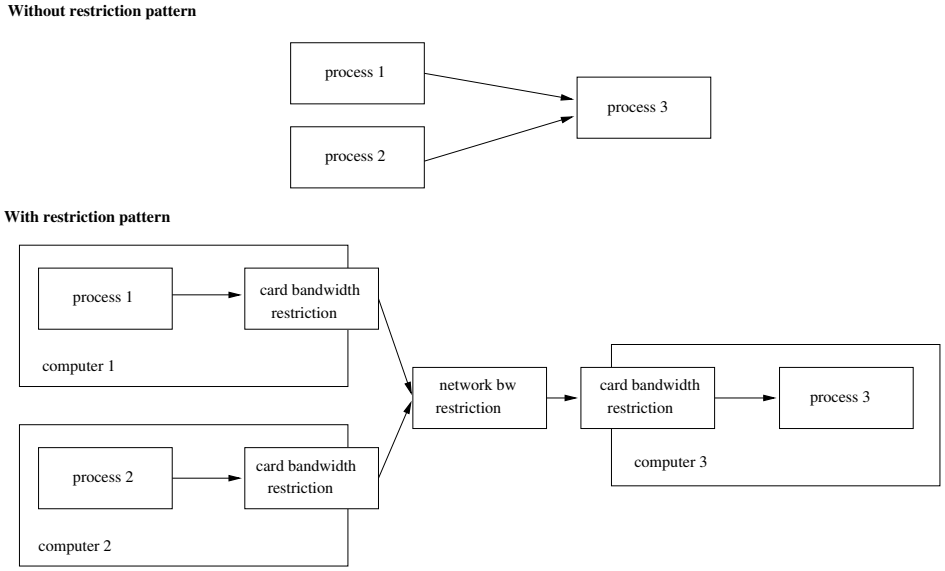


Figure 5.8: Example of the resource constraint pattern

need to be defined to avoid system overload. In order to *plug a constraint* at any point of the system, both in the inter-component communication and inside each of the components, a special simple component with only one process is defined. The constraint process implements the same API as the rest of the components of the system. A restriction about the bandwidth of the network that connects two components can be placed between them putting a transparent proxy process (decorator [GHJV94] design pattern) that implements this restriction and reduces the *options* answered from a component to the higher level, depending on the available resources at each moment. The resource constraint pattern, therefore, emulates resource limitations at the software level. As it needs to be able to understand the message API of the system, it can be seen as an extension of the trader design pattern, where the streaming options received from the connected subsystems are filtered depending on the maximum resources available and their current usage. In Fig. 5.8, an scenario where this pattern could be used is shown: specialized processes implementing network and card bandwidth constraints are used for limiting the amount of bandwidth available for the communication among three processes.

5.3.2 Internal protocol (message API)

As the system is based on components that can be plugged in very different ways, it is interesting to keep the internal protocol as simple as possible. For the basic functionality, four main messages are defined. All of them assume that the system architecture is a tree-like hierarchical structure, as will be explained later on in this chapter:

- *Lookup message.* A component (or an external application) sends this kind of

message to another component whenever it wants to check if streaming a concrete *Media Object* from that component (or another component depending on this one) is possible. In *Erlang*, the simplest version of this message would be composed by a tuple with three elements: {`lookup, MO, Profiles`}. The first element is an *Erlang atom* which indicates the message type. The second one is the *Media Object* identifier, unique inside the system. And the last element is the description of the non-empty list of qualities of the *Media Object* that are accepted. Converting the quality of the *Media Objects* on demand is too expensive both time and resources-wise, so they are stored in different sizes and qualities. It can be the case that only some of the qualities are available in the component, or that the resources available are not enough for streaming some of the qualities, so the sender component needs to specify all the qualities that it accepts.

- *Lookup answer message.* It is the normal answer sent back to the caller by a component when a *lookup message* is processed. In *Erlang*, the simplest version of this message is again a tuple, but now with only two elements: {`lookupAns, Options`}. The first element identifies again the message type and the second is a list of alternatives for streaming the *Media Object* that was referenced in the corresponding lookup message. If the list is empty, the component is notifying the caller that either the system is too loaded or the *Media Object* is not present. Each element in the list is another tuple, now with three elements: {`Profile, Cost, ProcId`}. `Profile` is one of the profiles contained in the lookup message. `ProcId` is the identifier of the concrete process (part of the component or the components that are *behind* the called component) that has the *Media Object* and enough resources for streaming it with the quality specified in `Profile`. Finally, `Cost` gives information about how *costly* is for the server to stream from a given process compared to streaming from the rest of them. For example, if the *Media Object* requested is available both in a tape component and a disk component, the cost is going to be higher if the *Media Object* is streamed from the tape. Also, if it is available in two disks with the same features, but one of them is already streaming other *Media Objects* (and therefore has more resources being used), the cost is going to be smaller for the disk that has less load. Cost values are used by the caller components later on in order to select from which of the processes to request the streaming. Depending on the concrete scheduling policies, cost values vary from simple numbers to more complex structures with information about the consequences of choosing each of the `Options`.

- *Play request message.* Once the *lookup message* is sent all the way through the architecture of the system, and the streaming options are collected back from the *lookup answer messages*, the caller components need to send a *Play request* if they want to start the actual streaming. In *Erlang*, a simple version of this request is another tuple of four elements: {`play, MO, Profile, SrcProcId, DstProcId`}. Apart from the message type, the `MO` indicates which *Media Object* should be streamed, the `Profile` specifies the desired quality, and the `SrcProcId` and `DstProcId` are the identifiers of the process

that should carry out the streaming and the process that is going to receive it, respectively. From the instant when the lookup message is answered to the moment when the play request is received, the available resources, or even the available *Media Objects* can vary; therefore, on the receiving of the play message, resources need to be checked again.

- *Play notification message.* It is the normal answer sent back to the caller by a component when a *play request message* is received. As we have explained, the resources need to be checked again, and it could even be the case that the *Media Object* have disappeared from the device controlled by the source process. Therefore, due to this and other technical reasons, the play request cannot always be satisfied. The play notification message has two possible forms, one produced when the streaming is set up, and the other when it is not possible. In the first case, a *transfer* can be created with two *pipes* connecting the source and the destination processes. The protocols used inside each of the *pipes* can be negotiated by the processes themselves or imposed by other components, depending on the concrete configuration of *VoDKA*. An special relation is created between the processes in the *transfer* and those handling information about resource usage, and when the streaming is finished, they are notified so that the resources can be actually freed.

Most of the processes involved in the scheduling subsystem communicate mainly using the four messages just described. This message API is, therefore, the one implemented by the processes that follow the *trader* design pattern inside *VoDKA*. The way these messages are forwarded and answered back through the system architecture defines the concrete configuration of *VoDKA* and how the components are connected.

Apart from the messages described, there are other ones related to the error messages, the process supervision, the negotiation and creation of the pipes and transfers, etc. As they are not really needed in order to understand the part of *VoDKA* that is relevant for this thesis, we will not go into detail with that messages.

But we still have not talked about advanced features present already in the *use-cases* we have seen: slot booking, VCR-facilities, and *Media Object* management, optimize resources for some *Media Objects* and response time estimations. Slot booking messages can be seen as extensions to play messages. The behaviour of the components is very similar and the resources are set to being in use, but the streaming is delayed. VCR-facilities are special messages that interact directly with the *transfer* process, and do not need to go through the scheduler subsystem. *Media Object* management is done using simple messages sent from some kind of high level software to the components taking care of the storage devices. Resource optimization can be seen as a special case of *Media Object* management where a set of selected *Media Objects* are moved up in the system hierarchy (we will see more on this in the next sections). Response time estimations can be added as part of the lookup answer message if that feature is offered by the concrete version of *VoDKA* that is in use.

5.3.3 Description of software components

In this section, we describe the main software components without mapping them to concrete nodes or hardware configurations.

A multi level architecture, based on the chain of responsibility [GHJV94] pattern and the common internal communication protocol described in the previous section for all the components of the system was chosen for the software architecture of the server. Each of the components acts as a black box with a clear functionality for the rest of the system, and with a well defined communication API. The components can be plugged together following different configurations, that will be discussed in the next sections.

The proposed architecture satisfies the needs imposed by all the system requirements explained in the previous chapter. Apparently incompatible requirements such as low response time and cheap storage capacity, are conciliated by specializing components. Each component gives answer to the demands of a given subset of the in theory contradictory requirements. Of course, the rest of the requirements we have described, those that are not conflictive in this sense, need also to be taken into account: this is the case of features like availability and reliability, or upgradability and maintainability. All of them have been also taken into account in the design, and the best features of the development platform that have been used to provide support for those complementary requirements, as we will explain later in this chapter.

The main high level components that cooperate inside *VoDKA* are:

- Streaming component: Its functionality is related to the protocol adaptation with the end user. It needs to be very fast (software is defined to be so and special hardware would normally be used for the deployment), but it does not need to be able to store a high amount of *Media Objects*. The communication between the components inside the server is done by using internal, *ad hoc*, communication protocols. These high level protocols are more adapted to the needs of the internal intercomponent communication than the general purpose TCP/IP protocol (which can still be optionally used in a lower level). The end user requests a given *Media Object* with a concrete quality (bandwidth use profile) and one of the typical streaming protocols (RTP, H.263, HTTP, etc.). Internally, the streaming component consists of three types of processes: the *streaming scheduler*; the *streaming group*; and *protocol frontends/streamers*. The different frontends for each of the protocols receive the *Media Object* requests from the user; for each request, they build a *lookup message* and send it to the *streaming group*, which follows-up the request to the *streaming scheduler*. The streaming scheduler defines the logic for the whole component: it knows how to contact the next level (another high level component normally acting as an intermediate cache) and sends the request, waiting for a *lookup answer message*. On the way back, the streaming scheduler could decide to choose among some of the obtained options or forward all of them to the user. Once the source is chosen, the *play request message* is sent following the same way through the system, as it was already explained in the previous section. Internally, all the process cooperating in the component are implemented using the design patterns explained: they are traders based on reflective

generic servers, and the internal structure of the component is a supervision tree. Monitors are used in order to extract information from the system, and resource constraint behaviours are introduced when the software needs to be aware of the underlying hardware limits. The scheduler process implements the design pattern with the same name.

- Cache component: The reason for configuring the server deployment using this component between two other components is to reduce the overall system requirements, by placing closer to the user the *Media Objects* used more often. The cache layer in the hierarchical architecture needs to have a reasonable speed, although the needs are relaxed by the streaming component, and it needs to have a larger capacity, for example, but still not as large as the next levels of the hierarchy. Internally, the cache component contains three main types of processes: the *cache scheduler*, the *cache group*, and the *cache drivers*. The *cache scheduler* receives the *lookup* and *play* messages from another component and implements the following protocol: it asks to the *cache group* about the availability of the *Media Object* inside the local cache of the component; if the *Media Object* is there, answers with a *lookup answer message* to the requesting component; and second in case the *Media Object* cannot be found locally, the request is forwarded to the next component. The *cache group* acts as a common facade for all the drivers. The *cache drivers* have all the implementation of the typical cache algorithms, and all the information about how (quality), where and which *Media Objects* are available. Again, all the process cooperating in the cache component follow the described design patterns: they are also traders based on reflective generic servers, and the internal structure of the component is a supervision tree which provides fault tolerance. Monitors and resource constraints are also used when needed inside the component. The scheduler process implements again the design pattern of the same name, being in charge of filtering the options if needed, taking into account the cost information as it was already explained in the message API section.

- Storage component: It is quite similar to the cache component in its internal structure, but its main goal is to be able to store a huge amount of information. As it is very difficult to conciliate the low cost requirement with the performance and high storage capacity, for this component the performance requirements are relaxed. Normally, this component is used in combination with the cache component, giving to specialized *levels* in the architecture: the cache component needs to have a smaller latency time, reducing that kind of requirements for the components connected to it (which is normally the case of the storage component). Therefore, the server global performance does not depend directly on this level in terms of load time, latency, throughput, etc. Internally, the component consists also of the same three types of processes: *storage scheduler*, *storage group*, and the *storage drivers*. As an example about how flexible the system configurations can be: one storage driver could be, in a complex scenario, a connection to another *VoD* server running in a different location, i.e. one server can act transparently as storage device for the other (another advantage of having a uniform message

API). Again, design patterns like traders, schedulers, monitors or resource constraints are used inside the component in order to reuse source code and simplify the internal architecture.

- Monitoring: apart from the internal monitoring that can be done inside each of the previously explained components, an specialized one in charge of receiving information from all the processes in the system is available in the *VoDKA* component library. The fact of having reflectiveness inside most of the low level processes, together with the frequent usage of the monitor and the event handler patterns, makes the information that can be extracted from the system very rich. On top of that facilities, high level interfaces, allowing the user to learn more about the system internals and giving real-time feedback, can be developed. This component is created using the monitoring pattern that has been already explained before.

Although we have included the resource constraint pattern as part of the high level components, used internally for modelling the underlying hardware limits, it could hypothetically be used between two of them. For example, if the cache and the storage are in different LANs and they are separated by a network link with a maximum bandwidth, the standard configuration design choice would be to place a process in the middle following the constraint pattern. However, an interesting idea is to standardize where those restrictions are placed, and a good option is to situate them always in the caller component, as the last process before the message is forwarded through the network. That way the architecture is homogenized and the message latency is reduced because the messages are only cross the network when needed.

Transfers are also a special kind of low level component (processes implementing patterns) that is not always inside the high level ones. Sometimes they can be created in order to transmit information inside one of the components (for example in a cache level for moving a *Media Object* between two devices), but in general they are used for data transmission between components all over the system. The usage of the transfer and their internal pair of pipes is, therefore, something transversal to the division of the software architecture in specialized components.

Sometimes a group of components work together implementing a concrete functionality. When deploying them, they are frequently started, stopped and configured as a whole subsystem, using some set of predefined interfaces. The generic *behaviour* can be abstracted, creating a design pattern for this kind of systems of subsystems, which would be parameterized by the concrete set of functions provided each time the pattern is used. The component would be called Application. In the case of *VoDKA*, depending on the concrete deployment, the whole system can be composed by one or several applications.

The components we have described and the different ways they can be combined, together with the selection of the right hardware, show already the solution the *VoDKA* development team came up with. In the next sections, more detailed information on how the components were actually implemented and how they are deployed will be shown.

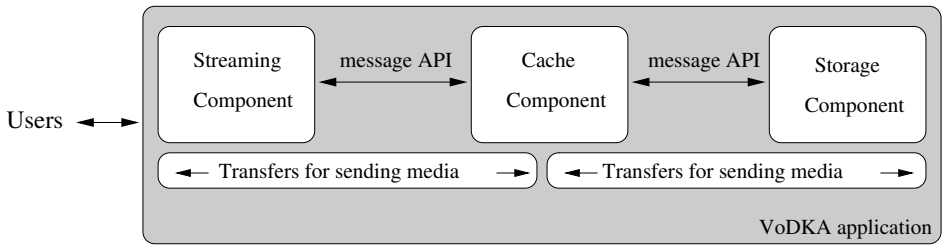


Figure 5.9: Simple configuration of the high-level *VoDKA* components

5.4 Process View

Everything in the message passing, process oriented, distributed philosophy followed in the *VoDKA* design is closely related to what in the object oriented world is called the *Process view*. Thus, it was not straightforward to determine what should be specified in this section. In the classical interpretation of the *4+1 Model*, the logical view includes the information about the classes participating in the system, and how they logically react with each other, while the process view talks about how concurrency, if existent, is included into the system, how many threads are executed at the same time, and which parts of the logical view participate in each of the threads. That distinction fits very well in data oriented software, where the logical view is conceptually very far away from the concrete number of threads that are used. On the other hand, in our case, the logical view already talked about process-oriented design patterns, low-level process components, and processes collaborating inside high level components. We have even described the message API as part of the main entities involved in *VoDKA*, therefore also part of the logical view.

Due to all this, we kept the process view simple and we concentrated in giving here some extra examples about how the system components cooperate.

The different components can be combined in several ways depending on the needs for each of the deployments. Each of these combinations are called *software architecture configurations* (or simply *configurations*) of the system. This feature of *VoDKA* closely relates to the adaptability requirement explained in the previous chapter.

A frequently used and quite simple configuration would be the one composed by one *storage component*, one or more cache components, and one streaming level, internally using different resource constraint subcomponents that would model the underlying hardware limitations. This basic software configuration can be seen in Fig. 5.9.

Depending on the concrete requirements (for example, the performance that is demanded for a giving configuration), one cache level could not be enough. In that case, an alternative solution is, instead of increasing the size or resources of the existent cache, or putting a new cache besides the other one, to create a hierarchy of cache components, that would also follow the delegation model. In that configuration, the first cache component would need to have more speed but less capacity and the last one would have requirements more closer to the ones in

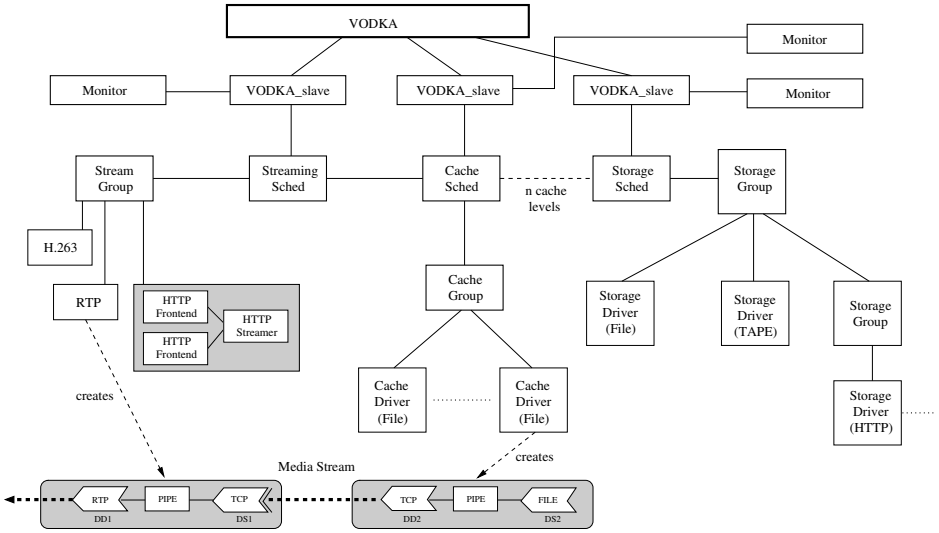


Figure 5.10: Processes in a linear configuration of *VoDKA*

the storage component.

This figure with the linear configuration can be extended so that these multi-level caches are introduced and, what is more interesting, the internal processes of each of the high-level components are detailed. The result can be seen in Fig. 5.10.

The upper part of the figure represents the supervision tree of the processes in charge of starting the system and providing fault tolerance. They start all the components and supervise them so that if any of the subsystems crashes, they can be restarted. Also, some monitors have been added, each of them receiving notifications with all the interesting information produced in each of the system components; monitors are also connected to the supervision tree so that fault tolerance is provided for the monitoring subsystem.

In the middle part of the figure, the internal processes of each of the three main kinds of components are shown. The streaming component offers to the user applications different kinds of front-ends (depending on the complexity, some of them can be composed by several processes). The cache component is in charge of several devices where the *Media Objects* are stored temporarily in order to increase the system performance. The storage component takes care of several devices where all the *Media Objects* present in the *VoD* server are stored permanently. One of this storage devices is, in fact, a *wrapper* for another *VoDKA* system which is accessed using the HTTP protocol.

Finally, in the bottom part of the figure, an example transfer composed by two pipes, where different protocols are used for reading, sending and receiving the media stream, is shown. In this case the pipes of the transfer are created by a process in the streaming component (the RTP protocol front-end), and a process in the cache component (the process controlling one of the file devices). In a running system, a lot of transfers will be transferring information through the server at the same time. All the processes involved in the transfers are eliminated once they finish their streaming tasks.

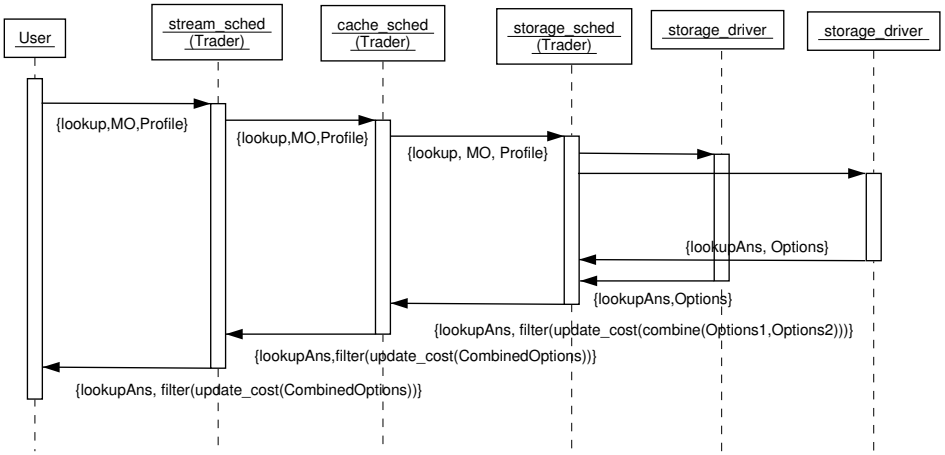


Figure 5.11: Example of message exchange in a linear configuration

Fig. 5.11 shows a sequence diagram where the messages described in the previous sections are sent through the *VoDKA* architecture configuration when the user requests information about the alternatives for visualizing a given *Media Object*.

As we will see in the following sections when looking into the different deployment scenarios, in most of the cases having only one storage component is not enough. The same happens with the streaming component: normally the system configuration has several of them connected to one or more high level cache components. The scalability of the system, one of the relevant requirements described in the previous chapter, can be satisfied adding new components to the system, together with the corresponding underlying hardware resources. But according to the description we did, the system needs to be also downward scalable, so that for very simple scenarios the number and complexity of the components can be reduced; this is satisfied because *VoDKA* can even work with a simpler configuration than the three standard levels: only a streaming component sending directly the media from the storage component.

5.5 Development View

In this section we explain more about the actual system implementation using *Erlang/OTP*, and the development process followed in the project. The goal is to complement the *4+1 Model* with some extra ideas that were not easy to include in the classical views.

All the system components and processes discussed in this chapter have been implemented in the *VoDKA* project using the *Erlang/OTP* platform, already described in Section 3.1.

Obviously, the approach selected for the analysis and design of *VoDKA*-being still quite general- fits very well with the philosophy behind a message passing distributed language like *Erlang* and its platform. The development team had *Erlang/OTP* always in mind as the target platform and therefore all the steps taken where addressed to that kind of development.

Although the design patterns presented in the previous sections are generic, they are easily implemented in a language like *Erlang*. In fact, some of them are offered by the OTP libraries as basic components that should be used in most of the systems developed with *Erlang* (e.g. the FSM pattern or the generic server), but others have been developed by the *VoDKA* team as extensions to those patterns, some of them being applicable outside the *VoDKA* system (i.e., reflective generic servers or transfers with pipes).

When the system development started, the idea was to use *Erlang* for the control subsystem, that is where the concurrency is handled and where the distributed high-level technology seems to fit better. A lower level language like C would be used for the critical parts, where performance is relevant. After the development of the first prototypes, *Erlang* resulted to be fast enough for satisfying the performance needs, avoiding the bottlenecks of the system to be in the software, without the need to do any special parts in lower level languages (still, some specialized modules, for example those related to embedded systems, are developed using low level languages, but they are not relevant for the overall reasoning about *Erlang* and its performance).

The system was developed as a group of *Erlang* components, each of them defined as a set of *Erlang* modules that implement different behaviours (predefined or *VoDKA* specific). In the code, the developers tried to take advantage of all the high level features of the declarative languages. The resulting software is quite easier to maintain compared to a similar application written in any lower level language.

Another interesting aspect is the use of *GNU/Linux* as the operating system for giving support to all the complex software architecture that we explain in this chapter. The fact of using *free software* allows the development team to fine tune the system up to the point of modifying or adapting the kernel to the performance needs of *VoDKA*, when needed.

VoDKA has been developed by a reduced team of developers following a *eXtreme Programming* approach, doing rapid prototyping and adding progressive new features doing fast development against a unique repository. One of the main features of *Erlang* is in fact the very fast prototyping, which is very adequate for this kind of projects where the system design is not completely clear from the beginning. In the case of *VoDKA*, all kind of redesigns were carried out in successive versions of the system.

5.6 Physical View

The *VoDKA* deployments -also called *system configurations*- map the software architecture configurations into different hardware architectures.

In the following chapters of the thesis, when using formal methods for extracting information about the system, we will take as input the software architecture. This can seem to be contradictory with what we explain here about the importance of the underlying hardware, but it is not. The *VoDKA* software is aware of the hardware limitations, that are included in the control system as internal constraints, as we have explained, so we will be able to extract interesting information only looking into the software side.

But we leave the system analysis for later. *VoDKA* has been deployed in different scenarios with heterogeneous networks and requirements, and here we just to give some representative examples.

5.6.1 *VoDKA* very simple deployment

This first deployment example is a direct mapping of a very simple software architecture with a streaming component, a storage component and without cache, into only one machine that acts as a server. Therefore in this example *VoDKA* is not distributed, although of course the software architecture is still very concurrent.

Fig. 5.12 shows an example about how the information transference from storage levels works until the actual streaming of the object in a simplified system configuration where the cache has been removed. In this case, the client request is received by an HTTP front-end, which defines the adaptation protocol required for a distribution of type *progressive download* over HTTP of a *Media Object*. The front-end interacts with its scheduler (*Streaming Sched*) through the group in which is integrated (*Sched Group*), and decides the way in which the video stream is actually going to be distributed (`DD1`, in this example, instantiated to an HTTP adaptation in a port negotiated with the client). The streaming level scheduler propagates the *Media Object* request to its successor in the responsibility chain [GHJV94], incorporating the protocol that the storage should use for the transference (`DD2`, in this example a TCP/IP communication). The successor, the storage level scheduler (*Storage Sched*), propagates the request towards an storage multiplexor (*Storage Group*), connected to different storage systems: a mounted file system (*File Storage Driver*), a tape robot (*Tape Storage Driver*). The scheduler mission is to decide which source is going to be used for obtaining the *Media Object* (in the example the *File Storage Driver*), building a *pipe* that connects that data source with the transference protocol suggested by the streaming scheduler, that creates a new *pipe* for taking the storage transference and sending it using the destination suggested by the HTTP adapter.

This very simple deployment shows that *VoDKA* can also be used in contexts where the system requirements are not very high, being therefore downward scalable. It is not difficult to see that if the requirements grow, a trivial solution would be to divide the deployment into two different machines placed in the same local network, one doing the streaming, with a faster CPU for the protocol adaptation and less storage capacity, and the other with less performance requirements and a higher number of disks and external storage devices attached.

5.6.2 *VoDKA* on the Borg cluster

In parallel to the development of the *VoDKA* system, the *LFCIA-MADS* group built a cluster called *Borg*. *Borg* is an approximation to the concept of a *Beowulf* cluster, a *GNU/Linux*-based, low cost, distributed system. The usage of Borg as one of the natural hardware architectures to deploy *VoDKA*, conditioned positively the software architecture features of *VoDKA*, offering advantages when comparing to the existent *VoD* commercial solutions. The distributed memory architecture complements itself perfectly with the message passing philosophy of *Erlang*. Although the *VoD* system runs on other architectures as well, a *GNU/Linux* cluster

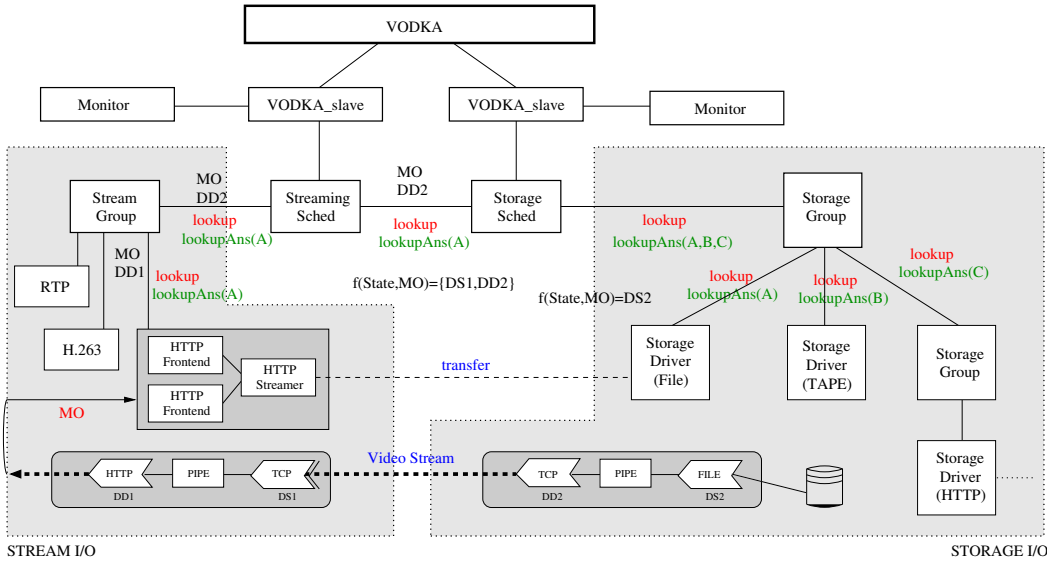


Figure 5.12: Simple deployment of *VoDKA* with only two components

provides an ideal environment.

There are several advantages derived from the utilization of this technology: the group had previous experience with distributed systems and clustering, that can be added to the wide experience existent in the *Open Source* community; the fact of the source code being available allows the modification of any part of the software for locating and correcting performance issues; the *Open Source* homogeneous license makes easier the legal treatment of the code; code developed for *GNU/Linux* is compatible, as it can easily be ported to other UNIX systems due to the respect of different standards; the performance is good; there are all kind of development tools available; and several hardware platforms are supported.

The *Borg* cluster (Fig. 5.13), used for the system deployment, was composed by 23 nodes plus a special front-end. Each of the nodes was a regular PC with two network cards (*Fast Ethernet*). The front-end was a dual Pentium with more memory and three Fast Ethernet cards, and acted as a gateway with the stations external to the cluster. Additionally, the front-end also worked as NFS server for the nodes.

The communication between the processors in the Beowulf cluster is carried by using standard UNIX network protocols. The Beowulf system was able to improve the communication bandwidth by routing packets through different Ethernet networks (using a well-known technique called channel bonding).

The cluster nodes are interconnected with 4 100Base-T switches, each of them with 24 ports, joined in groups of two by a 1 Gbit/s, defining in this way two different networks. This way, channel bonding can be used for increasing the network bandwidth, or one of the networks can be dedicated for administrative purposes, like NFS, using TCP/IP, meanwhile in the other network a lighter protocol is used for processes intercommunication.

The *Borg* generic architecture must be reconfigured in order to adapt it to the hierarchical three level structure (Fig. 5.14). As the massive storage level, a

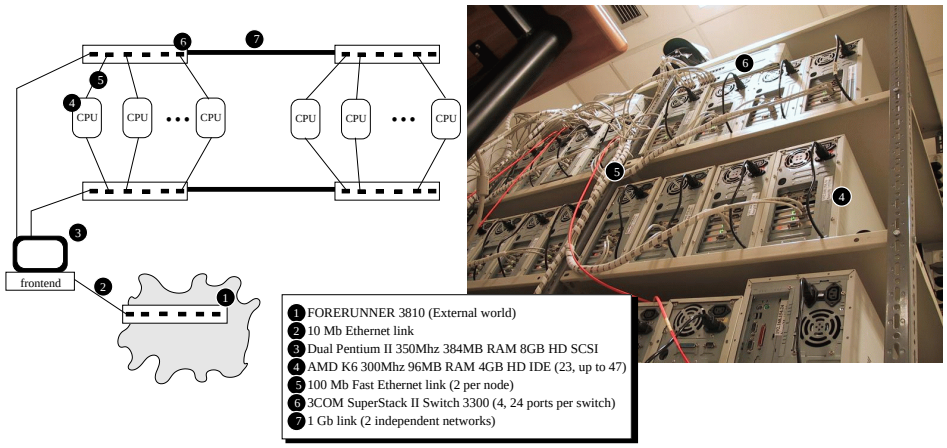


Figure 5.13: *Borg*, the *LFCIA-MADS*'s Beowulf cluster

specialized node with access to a tape robot can be used; the robot has access to potentially thousands of *Media Objects* but the latency of course is too big and needs to be reduced by an intermediate cache component. In this node, the storage software component controlling the tape device is deployed.

Working as a distributed *Media Object* cache, 23 nodes of the cluster are used; the *Media Object* is loaded into the cache from the tape robot on demand, and is resent to the streaming level from there; according to the cache policy algorithms, the *Media Objects* need to be replaced when there is no room for new ones. In each of the nodes, a cache software component taking care of the disk devices is deployed.

Finally, as streaming level, another specialized node is used, with higher memory requirements and gigabit Ethernet connection to the external world (the link of the *VoDKA* server with the users), as can be seen at the figure. The connections between the levels are done through the switches, and the front-end acts as the administration node. In this node, a streaming software component offering RTP protocol adaptation to the users of the server is deployed.

With this deployment, interesting results can be obtained when the network topology is not very complex and the goal is to be able to serve a big amount of concurrent users that are *close* to the whole *VoD* server configuration. In other cases, where the network topology is more complex, the deployment needs to be varied to fit into that topology; we will see an example of this in the next sections.

5.6.3 *VoDKA* on a faculty network

As an example of the *VoDKA* server configuration flexibility, Fig. 5.15 shows how responsibilities are distributed among the different nodes of the *Borg* cluster to give media services to the *LFCIA-MADS* laboratory.

- The massive storage is deployed in the node `borg25`, that also has an associated scheduler with two storage controllers (CD unit and tape robot with huge storage capacity).

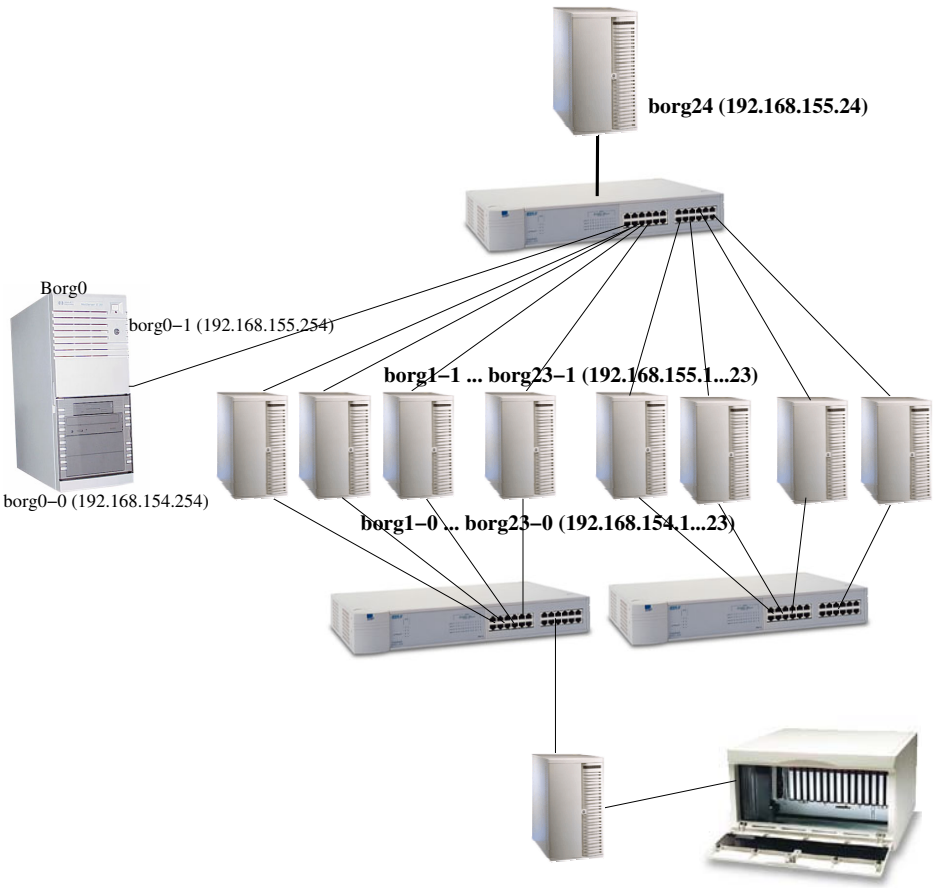


Figure 5.14: The *Borg* adapted to the hierarchical server architecture

- The storage scheduler is the successor of the cache scheduler, that is running in the node `borg24` in the responsibility chain. The cache scheduler uses as cache the aggregated bandwidth of the local cache controllers of the nodes `borg1`...`borg23`.
- The node `borg24` itself hosts an streaming scheduler whose successor is the cache scheduler, supporting a progressive download HTTP adapter, that gives video service to the *LFCIA-MADS* lab using the department switched 10Mbps network.
- The server, called `covas`, hosts a cache scheduler, whose successor is the `borg24` cache scheduler (two levels of cache), and a local cache controller. Besides, the server contains an streaming scheduler fed by the cache scheduler, and supporting an progressive download HTTP adapter, using the ATM adapter that is directly connected to the university backbone.
- `borg0`, the *Borg* cluster front-end, is used for the system monitorization.
- One of the nodes is a SPARCstation, while the rest of the nodes are x86 machines all of them running *GNU/Linux*. `covas` is a Sun UltraEnterprise

3000 running Solaris, an example of the portability of the server.

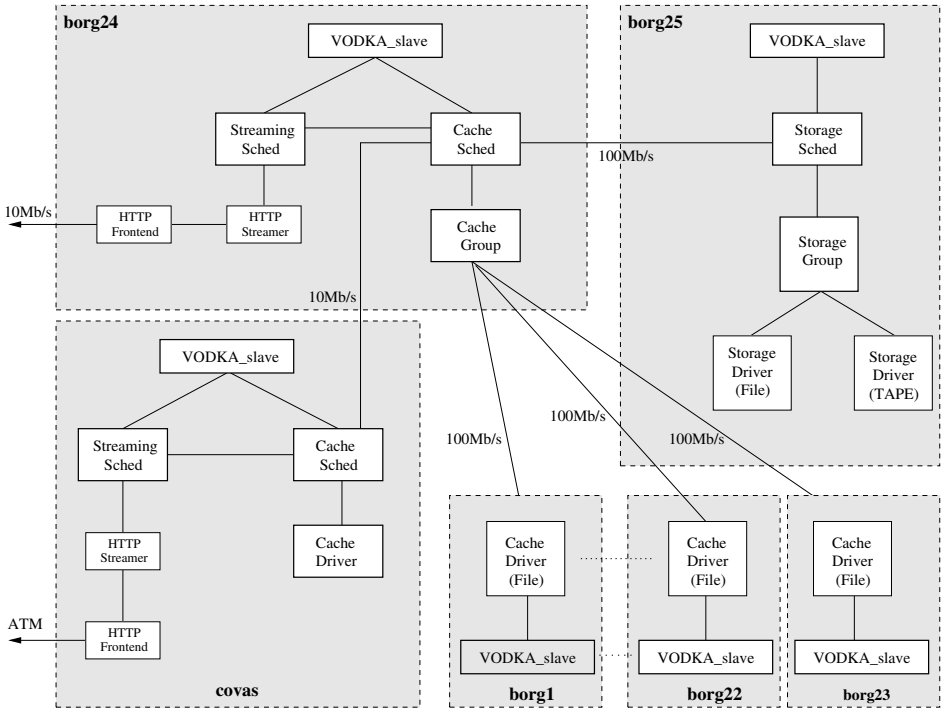


Figure 5.15: Deployment of *VoDKA* in a faculty network

This example shows already the flexibility of *VoDKA* and how it can be adapted to a faculty network with different subnetworks. Two streaming schedulers are used in order to improve the overall system performance and reduce the latency when the users request *Media Objects*.

5.6.4 *VoDKA* on a city cable network

The forth example shows a deployment in a typical cable network. The cable operator has a fiber optic network deployed over the main cities of the region. Each of the city networks has the following internal architecture: in order to reduce the cost and optimize the installation, a hierarchical network topology is used, where the bandwidth is smaller in the part of the network that is closer to the end user, and bigger closer to the central servers. The network is formed by three levels of rings: the primary network ring, connecting the primary network nodes in the topology, that are only a few, spread over the city network; the secondary rings, connecting each of the primary network nodes with their corresponding secondary network nodes, that are installed in each of the neighborhoods of the city; and the ring that connects those nodes with the tertiary level nodes, which are already in each of the buildings or small group of buildings.

The *VoDKA* software architecture can be adapted to obtain the maximum from that topology, as it is shown in Fig. 5.16. A massive storage component is placed in the central servers of the network, and it is replicated for fault tolerance reasons. In the primary nodes that connect with the next levels, a first level of

cache components is deployed. Each of those cache levels is a cluster of computers running distributed *Erlang* nodes controlling the different storage devices. In the tertiary level nodes, a second cache level is placed. Again, clusters of computers running the *VoDKA* cache component, dimensioned according to the number of users that are going to request *Media Objects* through that part of the network, are used. Finally, very close to the users, streaming components deployed in those nodes are in charge of doing the protocol adaptation for sending the actual stream to the end user applications.

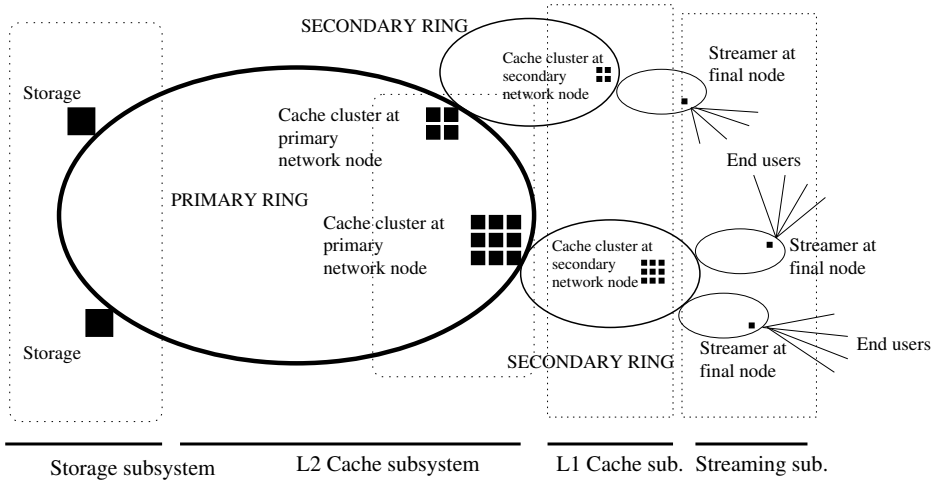


Figure 5.16: Deployment of *VoDKA* in a regional cable network

These configuration is a very good example of the flexibility of *VoDKA*, and how it can be adapted to a complex architecture where several storage, streaming and cache components are used.

5.7 Evolutions of the *VoDKA* architecture

We have shown the *4+1 views* of the *VoDKA* system. In this section, we discuss how these views have been evolved through the time, trying to show that the *VoDKA* architecture is constantly evolving, something that will influence the rest of this thesis.

The use of an architecture based on specialized levels was an initial design decision. After analyzing the requirements and reading the relevant research material, the only possibility to fulfill the goals of the system was to have three levels, one very fast for the protocol adaptation, one intermediate cache level, and the massive storage. In the first publications talking about the design of *VoDKA*, the architecture proposed in Fig. 5.17 was described. The configuration was fixed to this three software levels and to a given hardware deployment, and we were not talking still about specialized components, standard message API, or any kind of adaptability.

Soon it was clear that the flexibility was not enough. The solution was acceptable for simple deployments, where the network topology and the requirements where not complex. But, as we have shown when talking about the deployment of *VoDKA* in the metropolitan cable network, the number of software levels and

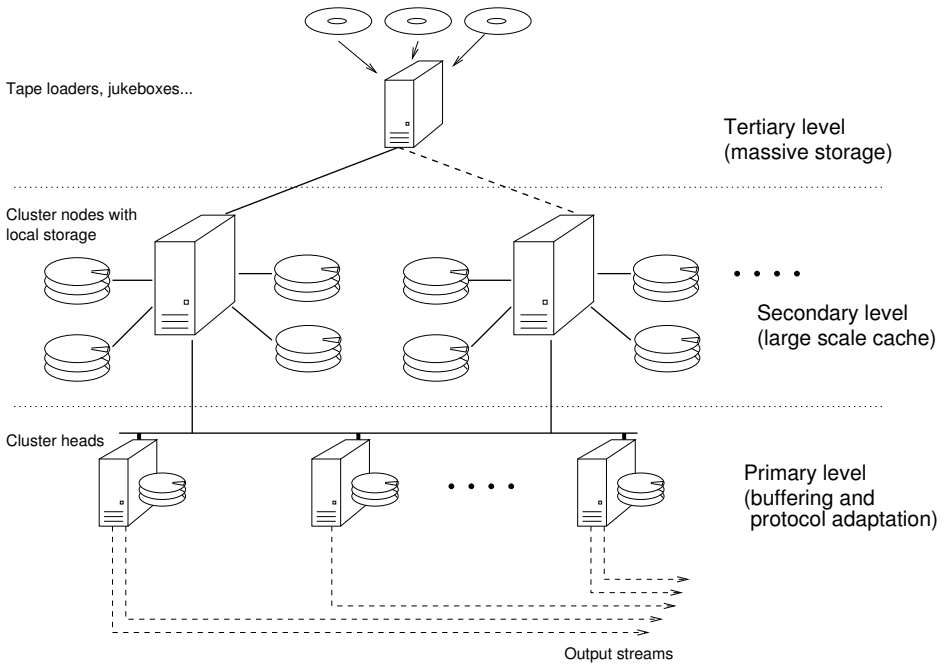


Figure 5.17: Original *VoDKA* design with a fixed 3-levels architecture

how they are mapped to a concrete hardware need to be adapted to the concrete needs of each scenario. As a consequence of this, the flexible component-based architecture was designed.

But the architecture flexibility is not the only thing evolving, there were also changes inside each of the components. The decorator based approach for modelling the underlying hardware resources has also been revised, and alternatives have been proposed in [PR03].

Given that restrictions in the proposal we have presented are simply decorators included in the responsibility chain, a problem can appear when a resource needs to be shared between two different chains of processes that are disjunct. There can also be problems when one resource needs to be used in two different points of a chain. Some workarounds can be used to try to solve this, but not for all the possible system configurations.

The alternative intra-component architecture, shown in Fig. 5.18 is based on a *component oriented constraint management*. Each of the system components (normally associated with a physical computer in the deployment) is going to have a *constraint manager process* that handles all the restrictions for that component. The constraint manager stores a resource table with the following information: {`resourceId, ResourceState, cost_function, check_avail, free_resources, alloc_resources`}. The tuple includes functions for updating the cost, for checking if a resource is free, allocating resources to a process and freeing, and a complex structure with the state of the resource usage in that node. This information (data and functions) can be stored in a distributed database, helping this way to obtain the fault tolerance for the server.

Each of the processes inside the component (storage scheduler and device drivers)

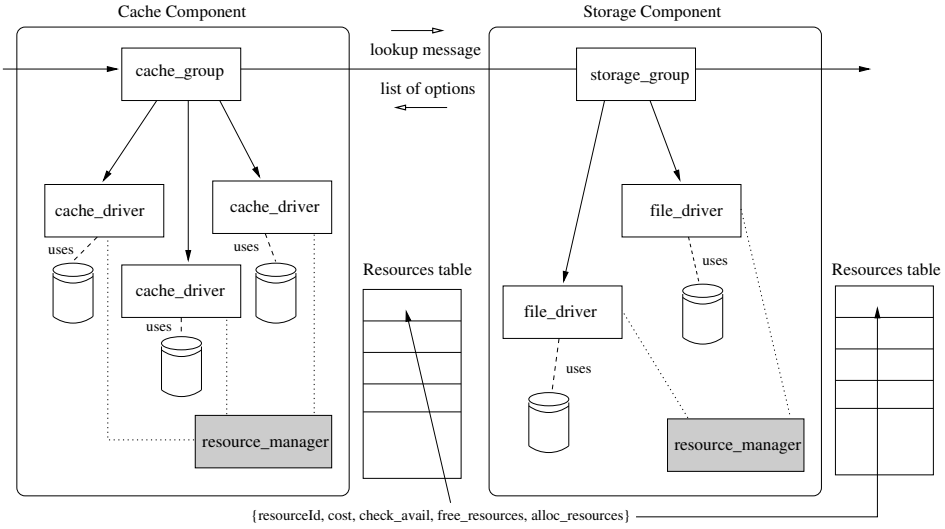


Figure 5.18: *VoDKA* components with the resource modelling process

is associated with a list of resources that are used by that process. For any checking or resource booking, the devices or intermediate traders/schedulers have to communicate with the constraint manager of their component. As the data is centralized, the restriction composition is easy to implement, and more complex restriction configurations are possible.

In order to implement this new architecture, a modification of the `trader` was proposed, where the state is extended to handle the resource manager and the list of used resources. A new message needs to be managed by the `handle_call` in order to be able to free the resources used when a transmission actually ends.

Summarizing, as shown by these examples, the architecture of the *VoDKA* project is very dynamic and evolves constantly at different levels.

Chapter 6

Lessons learned from *VoDKA*

Contents

In this chapter, we talk about the lessons learned during the development of the project. We will see how the proposed architecture meets the system requirements, how the *VoDKA* system compares with other solutions both in the academic and industrial world, the advantages of using *Erlang* technologies, and other ideas related to the future work. The goal is to explain which aspects of the system are more difficult to analyze, understand and improve.

6.1 *VoDKA* architecture evaluation against requirements

In this section, we discuss how the proposed system addresses the requirements described in Section 4.2. Most of these ideas have already been mentioned through the previous sections of this chapter, but we include them here organized per requirement:

- Huge storage capacity. The fact of using a hierarchical component-based architecture where the components are specialized introduced the possibility of using massive storage components without penalizing too much the overall system performance. In the previous chapter we have seen deployment examples where even a tape robot, with a very big latency, was used. Therefore, using several distributed storage components that are normally connected to the user using at least an intermediate cache level, the total storage of the system can satisfy the needs of, for example, a metropolitan cable network with several thousands of users.

- Large amount of concurrent users. The solution proposed is able to handle a big amount of users because of the specialization and the concurrency. Again, the use of cache levels increases the system performance, and the streaming levels, in charge of the protocol adaptation, are deployed using very fast

machines that are able to handle several streams at the same time. The architecture can be scaled up so that the number of streaming components is adapted to the requirements of each deployment.

- High bandwidth. This requirement is very related to the previous one and it is satisfied in the same way. Initially, the *VoDKA* development team expected the software to be the bottleneck, at least in some of the levels. Some of the low level parts were planned to be rewritten after implementing the first prototypes, using languages like C, but the conclusions of the first tests answered the opposite. *Erlang* was fast enough for this system and the limits of the architecture where more in the hardware side. Anyway, as for the previous requirement, scaling up the system, *VoDKA* is able to handle the required high bandwidth.

- Low response time. The main feature of *VoDKA* introduced for reducing the response time is the use of hierarchical cache levels, when needed. The fact of having the *Media Object closer* to the user makes possible to start the streaming earlier. Also, techniques for starting the streaming to the user before the whole *Media Object* has been received from other levels of the system architecture are used, in order to reduce more the time passed from the user request to the moment where the first part of the *Media Object* is received from the server. Optimization techniques for having better performance for the most popular content or even management decisions for increasing the popularity of the content with better performance are very useful when trying to reduce the average response time.

- Availability and reliability. We have explained that the supervision tree pattern is used in order to provide fault tolerance to the system architecture. If one of the processes inside one of the components crashes, its supervisor tries to restart it; if it is too critical, it could be that a big part or even the whole component needs to be restarted, but, in any case, after doing that, the system can continue working. The feature of continuing working when a hardware or software failure happens is deeply supported by using *Erlang/OTP*. The language and the platform are oriented to the development of distributed fault-tolerant systems, and *VoDKA* takes advantage the useful constructs, design patterns and libraries that are part of *Erlang*.

- Upgradability and maintainability. *Erlang* has built-in mechanisms for updating the system without needing to stop it; therefore, the *VoDKA* system is very easy to upgrade, just putting the right versions of the call-back modules in the right places modifies the system behaviour immediately. Several versions of the same code can be used at the same time, distinguishing which one is used by the way it is invoked. Maintainability is increased by the fact of using a high level declarative language, where the number of lines of code is reduced considerably compared to languages like C. The proposed architecture, where design patterns are used constantly, code is reused whenever it is possible, the message API is simple and homogeneous, and each component kind encapsulates its internal implementation, was designed to be easy to maintain.

- Experience-based automatic optimization. The capacity to learn from the past activity in order to improve the performance of the system is a quite general goal that can be partially satisfied with just small optimizations, but potentially is itself a whole research problem. Inside *VoDKA*, the system statistics, together with the values of the internal costs, and the different schedule processes, can be combined to do simple optimizations, like moving the recently most popular content to the upper levels of the caches, or pre-loading content that is known for being popular at given timetables. In any case, this concrete requirement can be still described as being work in progress and future work in the context of the *VoDKA* project.

- Scalability. The *VoDKA* system is very scalable. *Erlang* itself is a technology where distributing a system that is designed for one machine is almost a trivial task. With *VoDKA*, we take advantage of this, both at the process and at the component level. Inside a given component, more hardware and software resources can easily be added in order to improve the component performance. At component level, we have seen in the previous chapter, with the example deployments, that extra requirements can be satisfied adding to the architecture new components, running on top of *GNU/Linux* clusters. Besides, as seen in the simplest configuration we have shown, running just the two basic components in only one machine, the architecture is also downward scalable.

- Variability. Is the *VoDKA* system adaptable to the underlying network topology? Again, some of the deployments we have described show that the answer is positive. In the case of a metropolitan network with heterogeneous parts where bandwidth and maximum number of connections are different, the components can be dimensioned and spread all over the network topology so that the maximum performance is obtained. Streaming components can be moved to positions that are very close to the end-user, different levels of cache components can be used in the intermediate nodes, and massive storages can be placed in the central services of the network. Besides, the system can serve contents to applications demanding all kind of end-user protocols (some front-ends are implemented already, and more could be added easily due to the clean architecture), being also variable in that sense.

- Low (affordable) cost. Due to the use of *Open Source* technologies in all the levels of the software architecture, the cost of the system is reduced (no expensive licenses needed to be acquired for the development). The fact of using commodity computers for running the components, as seen in the *Beowulf GNU/Linux* cluster we presented in the previous chapter, and some variations of it that were used for different system configurations, is also an advantage when comparing the cost with those of the video servers that utilize specialized very powerful hardware to satisfy the rest of the *VoD* server requirements. Other features like maintainability or variability also reduce the overall cost of the system.

As seen above, most of the initial requirements were addressed by the *VoDKA* analysis, design and implementation.

6.2 Relation with other solutions

Although *VoDKA* architecture is quite innovative, it combines several ideas taken from previous research articles and from some industrial products. It is interesting to know how the server compares to those available products. In this section, a comparison with other options is discussed, pointing out both the ideas in common with *VoDKA* and the main differences.

If we look into the classification we did in Sect. 4.3, *VoDKA* should be included nowadays together with the enterprise solutions that are focused in LAN or WAN. Although it started as a research project, the implementation of the system has matured through the years and it is currently offered as a professional solution. The design of the architecture is focused in fast networks with each of the pieces of content have high quality, so its goals and therefor its architecture is very different to those.

On the other hand, there are some solutions that have inspired *VoDKA*.

The Apple Darwin Straming Server is also an efficient RTP streaming server, however, it does not include distribution facilities and it is not clusterable. Therefore, distributed massive storages, for example, are not supported.

Kasenna MediaBase, an evolution of the SGI's WebForce MediaBase, shares common features with the presented *VoDKA* design (it is modular, separates acquisition, distribution and streaming functions, and it is based on UNIX concepts), but has a lesser flexibility and adaptation capacity than *VoDKA*: its architecture is more fixed.

Probably the bigger inspiration of *VoDKA* is described in [CT97a], where a hierarchical solution for building multimedia servers is analyzed. This solution was based on specialized levels for streaming, caching content, and storing a big amount of *Media Objects*. However, although the ideas of the architecture where very interesting, and were adopted by *VoDKA*, the solution was less flexible and component based, and therefore less variable and adaptable to the needs of each concrete deployments. The *VoDKA* architecture can be seen as an evolution of what was presented in that article.

In general, there are some features of *VoDKA* that make it very different from the rest of the solutions. One of them is the use of *Erlang* and its related philosophy for creating fault-tolerant massive concurrent systems. Another important one is that instead of proposing a fixed optimal architecture, *VoDKA* provides components and leaves for each of the deployments the decision of which concrete architecture to use. The use of *Open Source* technologies and commodity hardware is also a very innovative feature.

Comparing performances is very hard to do. The access to results about the rest of the existent proprietary solutions is very difficult, and obtaining them by carrying out experiments requires a huge investment. Also, as *VoDKA* has very different possible configurations, and it is a very scalable system, the comparisons should be done between each of the systems and *one* of the possible *VoDKA* configurations. Selecting the right configuration to compare with is not trivial, and could have a lot of influence in the results obtained. Some performance results of selected *VoDKA* configurations are shown in [GBF05]. Also, a performance evaluation of *VoDKA* using queue theory for simulation is described in [VGM+00].

6.3 *Erlang/OTP* and *VoDKA*

In this section, we will try to analyze the advantages and disadvantages of using *Open Source Erlang/OTP* as the technology selected for developing *VoDKA*.

The used language, *Erlang*, has been designed and used in Ericsson for programming distributed control systems. The combination of the functional paradigm and parallel computing gives a declarative language, without side effects (excluding those that are needed for interacting with the low-level modules and the hardware), and with a high level of expressiveness, abstraction and ease of prototyping.

Erlang is specially suitable for distributed, fault tolerant, soft real-time systems like *VoDKA*. It is a language based on asynchronous message passing, transparent transference of values, and higher order communications, that has the capacity of supporting a high number of concurrent processes.

The language is suited for the development of distributed systems, permits the transparent location of processes in different nodes. It also includes primitives for the support of fault tolerance and provides facilities for the replacement of code without having to stop the system. All these features have been used in all the *Erlang* modules included in the *VoDKA* implementation.

The *VoDKA* implementation also uses extensively the libraries and distributed design patterns of the Open Telecom Platform (OTP), including generic servers, supervision mechanisms. This is probably one of the key features in the success of the project, and it will be shown as one of the more helpful ones in the next part of the thesis when trying to analyze or transform *Erlang* source code with different kinds of tools.

VoDKA uses also the *Erlang/OTP* distributed database (Mnesia) with location transparency, fragmentation, replication, and integration with the language, and a lot of useful integration libraries. SNMP, the Inets HTTP server, the SASL support libraries, the EVA and MESH alarm and measurement handling applications, and Mnesia are used by the monitoring subsystem. The C interface, the TCP and UDP libraries and others are extensively used in the I/O and streaming layer.

There are also additional modules in development (LDAP administrative interface, user application gateway) that make extensive use of ASN.1, and the Java interface, among others.

Erlang/OTP not only provides many useful libraries and applications; there is also a rather homogeneous philosophy underlying the platform,. A high degree of reusability and high programmer efficiency are also encouraged and made possible.

The fast prototyping that *Erlang* provided was also very useful and allowed to combine the research in the first stages with the implementation of the first prototypes, learning from them and doing changes in the system architecture.

Summarizing, all the features of the selected language fitted perfectly well with the system requirements and the designed architecture, and played a major role in the process of creating a successful *VoD* server fulfilling the expectations existent at the beginning of the project.

Of course, the project team found also some disadvantages in the language; the most important ones are:

- Lack of type system: a big set of the errors found during the system development could be avoid (or its solution could be simplified) with the help

of a type system for the language. This has a bigger influence in massively concurrent systems, where debugging is always by definition very complex.

Trying to give solutions, in the latest years some tools for helping in the debugging of *Erlang* systems have appeared. Some of them are based in code analysis, others more oriented to tracing the execution, and finally more ambitious ones that make use of formal methods tools and techniques. We will see much more on this in the second part of this thesis, when talking about how to analyze the *VoDKA* system for automatically extracting interesting performance information.

- Lack of module system: in a real system, with a quite big set of modules, a good module system makes easier the maintenance of the project evolution. Some proposals have been made in order to solve this problem in the language, and we found that they would be a really good contribution to the improvement of the Erlang/OTP platform.

- Small coverage: despite of the exponential grow of the Erlang user and developer community, it seems clear that the language is still not one of the best known ones, and this makes the task of finding good programmers harder.

- OO-world adaptation to Erlang nature: there is a quite broad work in the definition of design patterns in the object oriented programming. When designing the system, several tools (e.g.: UML) and concepts (e.g.: design patterns, composition, inheritance) from the object oriented world were used. Although most of the times translating these concepts into a process oriented approach is possible, this task is not always trivial. An example of a object oriented concept that is not easy to translate to Erlang is object inheritance.

6.4 Conclusions and future research

We have shown the design and implementation details of *VoDKA*. *VoDKA* uses innovative concepts in several parts of its architecture, and has shown itself to be a very good solution that beats some of the alternatives in several ways. The sever mets all the basic *VoD* requirements and adds three more that are key features of *VoDKA* and are not present in most of the alternative solutions: the adaptability to different protocols and underlying topologies; the downward and upward scalability; and the affordable cost both from the software and the hardware point of view.

The solution proposed uses *Erlang* not only as the development platform but as a technology that supports the philosophical approach, based o concepts like fast prototyping, massive concurrency and the fault tolerance.

Future implementation work is mainly focused in two aspects:

- Adding supporting for all the recent streaming protocols and media formats. It is important in order to maintain the system as a competitive alternative in the industrial world to support all the formats that keep appearing.

- Creating management and user-oriented applications for taking advantage of the server. In this thesis we focus in the study of the server part, but

on top of it, several vertical applications need to be constructed. Management applications are used by the companies running the *VoDKA* server in order to manage, for example, the financial information. The user-oriented applications are very heterogeneous and would be adapted to the needs of each scenario, like hotel *VoD* applications, museums *VoD* applications, or the video services that are starting to be used in the faculties for announcing news.

On the other hand, future research work is oriented to increase the intelligence of the system:

- More automatic auto-design. We have already said that among the requirements specified for the server, the one that is still less fulfilled is the experience-based optimization. Sophisticated algorithms could increase the performance of the server by doing the right *Media Object movements* before the majority of the user requests are received. More research would need to be done in order to take real advantage of this potential feature.
- Better architecture and new components. As we have seen, the architecture of the server is constantly adapted in order to satisfy the needs of new deployments being always *backwards compatible* and keeping all the previous components. Research should be done in order to learn more about the architecture and improve it even more. The next part of the thesis is an effort in that line of research.
- More code reusability and more sophisticated design patterns. Even thought the number of design patterns contributed by the *VoDKA* development team to the *Erlang* community is already considerable, in some cases there is still room for generalizing some other parts of the server and creating new pieces of generic code that can be used everywhere without the need of duplicating them. Also, studying how generally applicable are some patterns, like the *trader*, the *scheduler* or the *resource restriction*, is still subject of further research.

In this part of the thesis, we have learned the following key things about the *VoDKA* system:

- The initial requirements where complex and therefore the solution had to be very ambitious and flexible in order to fulfill them all. Better tools for checking how well the requirements are satisfied are required.
- *VoDKA* is a solution comparable with those found both in the academic world and the industrial world, which has some key features that make it a more suitable alternative for some scenarios. Better performance comparisons would enrich the knowledge about the system.
- *Erlang/OTP* and the distributed message passing approach where key features in making *VoDKA* a successful project. However, some of the disadvantages of using *Erlang* have been also described. Those disadvantages suggest the usage of advanced tools for reducing the amount of errors introduced by the developers in the programs.

- The architecture of the system evolves constantly and is under a continuous process of adaptation and enrichment. Tools for learning more about the architecture and for improving it are interesting for the development team.
- *VoDKA* is a successful project but still needs a lot to be done, especially in the process of learning about the architecture.

We have seen that debugging is difficult. The architecture is complex and it constantly evolves. Alternatives like simulation or testing have some problems. *Erlang* is a high level declarative language and using formal methods for analyzing it is a pretty reasonable way of advancing.

Using formal methods for enriching the engineering process in order to increase the system quality was a very welcome initiative inside the *VoDKA* project.

We will motivate the usage of formal methods for the *VoDKA* project and explain the tools, method and results obtained in the next part of the thesis.

Part III

Using formal methods for improving *VoDKA*

Based on the experience of developing *VoDKA*, and in parallel with the evolution of the system, a study has been carried out in order to find out how formal methods could help in the goal of improving distributed systems design and implementation. In this part of the thesis, composed by three chapters, we present the proposals, tools, methods and results of that study.

In the first chapter, the use of formal methods in the context of the *VoDKA* project is motivated. The existent alternatives are considered and discussed from different points of view, and the approach that has been selected is introduced. Questions that we try to answer here include why, when and how can formal methods be used for *VoDKA*, being always clear with the disadvantages and limitations of the approach.

In the second chapter, the tools that have been selected or developed for applying formal methods to *VoDKA* are described. For the tools developed the internal details are given; for the external ones that have been used, a short description with references is given, together with an explanation on which parts of this tools have been useful for the purposes of our research.

Finally, the third chapter shows the details of the method we have followed in order to analyze *VoDKA* using tools and techniques from the area of formal methods. The method is illustrated with several experiments that have been carried out for different configurations of the system. The results are provided and discussed, and then we conclude and point out the main research paths that have been opened.

Chapter 7

Formal methods for the *VoDKA* project

Contents

The purpose of this chapter is to put our research in context. There are a lot of different approaches for using formal methods in complex systems. During the last decades, different tools, technologies and methodologies have been proposed. Some of them are more theoretical, others have a more practical approach; some of them are isolated tools and others have been described as part of a complete software development methodology. In some cases the users of the tool are supposed to be developers, in other cases they are the designers or system architects. Where in this complex picture is our proposed approach located?

In the following sections we try to give an answer to that question. We talk about the advantages and disadvantages of using formal methods; we discuss the existent alternatives explaining the main differences among them; we describe the different possibilities in the software development where this kind of tools and methods can be used; we then elaborate on the skills needed for using the different formal tools, languages and technologies; and finally we discuss limitations and things that cannot be done following this path.

In all the cases, inside each of the sections, we go from a more general discussion on the existent alternatives to present and justify the approach we have selected for the *VoDKA* project.

The content of each of the sections refers to the chapter title, trying to give answer to one question that relates to that title: *why* do we use formal methods for *VoDKA*?, *what* methods and tools do we propose to use in the context of the *VoDKA* project?, and so on.

7.1 *Why*: advantages versus disadvantages

Although there are a lot of different approaches, tools, methods and goals, the most common answer to *why* to use them is that they claim to provide quality. We needed high quality for the *VoDKA* project, therefore, it is interesting to see what formal methods can offer us.

VoDKA is an example of software getting more and more sophisticated as the project evolves. The development of this kind of sophisticated systems that run on top of hardware that is each time more complex, requires at the same time more advanced tools. New languages are developed, new software development processes and new analysis and design methodologies are proposed, but this has shown not to be enough: software errors are still present and maintenance costs are claimed to be the biggest part of the software life cycle costs [Erl00]. In a society where software is present everywhere, a lot of the cases being responsible for critical services, we consider that the quality of the systems should be a big issue.

The approaches based on formal methods, that make use of a deep mathematical background, have been present since the first steps of the computer technology. However, although for the case of the hardware or the communication protocols they have been widely used in industry, their use for complex software has always been considered too costly. Therefore, it has traditionally been relegated to academia, where all kind of proposals have appeared, most of them without great practical success.

However, the more complex the systems are, the more difficult it is to obtain the desired quality level: errors are more easily introduced in complex developments. In these cases, the relative cost of formal methods can be acceptable in the context of the project development. This means that we should be able to use formal methods cost effectively to achieve a certain quality level, when systems get more complex and this level can no longer be achieved by more traditional methods.

Also, there is always another way of reducing the cost, lowering the level of use of formal methods, as it was described by Rushby [Rus93] using the concept *levels of rigor*, that go from not using formal methods to *fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking.*

But what do we mean by increasing the quality? ISO 8402 (1986) defined quality as "the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs". Later, in 1994 it was redefined as "the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs". Pressman [Pre97] defines quality as the conformance to explicitly stated functional and performance requirements; the explicitly documented development standards; and the implicit characteristics that are expected of all professionally developed software.

Quality can be seen as a complex concept including things like increasing the confidence on the correctness of the system, revealing ambiguities, detecting design flaws, finding some kind of incompleteness or inconsistencies, making the system easier to maintain, reducing the number of errors at different levels, increasing the knowledge about the system and its performance, and so on. Depending on each project, one or several of these quality aspects are going to have higher priority.

Selecting an approach adapted to the concrete goals is one of the key decisions for getting something useful from formal methods.

In the case of *VoDKA*, we are studying a system with a complex and flexible architecture. Analyzing the requirements described in Chapter 4, we were interested in building a system and using formal methods in parallel for increasing its quality. The architecture of that system was shown in Chapter 5, and the conclusions after several years of development pointed out in Chapter 6, support that our approach of using formal methods was indeed interesting. Following Pressman's definition, we concentrate specially in the first aspect: the conformance to explicitly stated functional and performance requirements. As *VoDKA* is going to be used for offering services that are similar to TV, users expect the same quality of service which has traditionally being very high. Examples of information we want to look into are system capacity, system bottlenecks, more knowledge on the architecture, or feedback on possible design or configuration changes.

VoDKA is constructed as a set of flexible components that can be plugged to each other in many different ways, giving place to what we call *VoDKA configurations*. In the development team, there was a strong focus since the beginning of the project in increasing the knowledge on the performance of the configurations of different versions of the *VoDKA* components. This includes detecting design problems, and finding errors, but is more focused on the automatic extraction of software architecture information from each version of the implementation.

Besides, due to the changing nature of *VoDKA*, there was low priority on updating the documentation of the system. The development team followed always an agile methodology, with rapid prototyping and adding new features to the system in a progressive way. All these circumstances conditioned the kind of approach selected for using formal methods, that needed be based on the direct analysis of the source code (the most updated system specification available).

VoDKA is an example of resources-aware software: the components are aware of the limits the hardware impose to them, in order to work properly. If a *Media Object* needs to be streamed, the components check first if the available resources (disk capacity, network bandwidth, disk to memory bandwidth or whatever other resource is involved) are enough, and only then an answer is returned. This is not an exclusive feature of *VoDKA*: it is present in a family of systems where capacity is normally an issue and load balancing algorithms are spread all over the system.

For these kind of distributed systems, formal methods was a promising way of increasing the quality, and they have been using with success in industry [CGR95, HLS+02, BH06, CGR93]. As we will explain, we detected that it was possible to come up with a method which cost was acceptable compared to the overall project effort. It could be applied in the context of the *VoDKA* project, and the results of that method would fit perfectly in the need of knowledge about the system performance and that would pay the effort.

7.2 *What*: methods and tools

Once we decided that we wanted to use formal methods for extracting information from resources-aware distributed systems, the next thing was to select the methods and tools we were going to use.

The alternative we have selected inside formal methods is model checking: generating the state space directly from the *Erlang* source code, and then verifying interesting properties in order to obtain relevant system information. The goal of starting directly from the source code is to avoid having to write models by hand. The approach has some natural limitations because the state space can be infinite, but we will explain how we take advantage of the *VoDKA* design patterns in order make the model smaller. We cannot verify everything using model checking, but we have come up with a method for doing a kind of stress testing of an abstraction of the *Erlang* source code that can obtain interesting and useful results.

Hereafter, we will discuss the differences between model checking and the rest of the alternatives.

The first alternative to take into account, probably because it has traditionally been seen as less complex and therefore less costly, is testing. Testing consists in the definition of a set of test cases that can be used for finding problems in a running system or in any of its components. Testing itself is not a formal method, but research is being carried out in combining it with formal approaches for generating automatically the test cases from logical properties. An example of this kind of initiatives is QuickCheck [CH00, AH03]. Testing can be used for units and components. This is basically testing for functionality. However, on the system level, it gets a bit tricky when it comes to testing. There are many possible configurations and when we are interested in functional, but in particular non-functional requirements of these configurations, we have to actually build them in order to test properties. In order to get the results one needs to have a system running, and in the case of *VoDKA* that means configuring a network, placing the movies, and simulating the behaviour of many concurrent real users that would request different *Media Objects*. In a very changing project, the effort of performing the testing steps each time a new release is produced, can be too time-consuming. Testing, however, has other advantages, like working on the real system and not on an abstraction, which can potentially give more real results.

Another approach for analyzing distributed systems is the use of simulation tools, frequently based on queue theory and statistic modelling, and normally used for non-functional requirements. This approach usually consists in creating a model from scratch during the design of the system, in order to be able to understand better the architecture and tune the configuration. Once the model is considered to be correct, some of these tools have facilities for directly generating part of the source code for a set of target languages. The simulation approach looks into the performance when the system has average load. However, creating the model is normally time-consuming and it does not adapt very well to very changing systems like *VoDKA*. During the development of *VoDKA*, some research [VGM+00] using Esterel/QNAP was carried out as a proof of concept, with the limitations and advantages commented above.

Theorem proving would be another alternative. It consists on reasoning about the system step by step in a mathematical way, trying to verify functional properties or find possible errors. The approach is very powerful, and in theory any kind of property can be verified for all kind of complex systems. There is even a tool, EVT [Fre01], developed for doing this kind of reasoning about *Erlang* systems. In practice, the effort required is very big, and the process is not automatic, requiring

human collaboration. In the context of *VoDKA*, where we wanted to help the development team with a very changing project, it seemed not to be the best option to use theorem proving from the very beginning. It could, however, be something to consider once the system is stable and mature, in order to certify some critical part of the components.

Although we have selected model checking, we think that all the alternatives in fact complement each other, and we could use them in different stages of the software development process. Some authors, for example, point out that formal methods like model checking and theorem proving can help in the process of learning about a system in order to make better test cases for testing.

With the model checking approach we will extract information about the system performance, which is very related to the quality of service offered by *VoDKA*. Therefore, this supports the goal explained in the previous section of *why* using formal methods.

7.3 *When* and *how* in the software development process

We now have decided to use model checking for increasing the knowledge of the system extracting performance information. Next question is related to *when* should this model checking be carried out inside the development process.

In general, some of the tools and methods from the area of formal methods are presented as isolated tools, not closely related to the more common software development processes and their different stages. In other cases it is implicit that they are used first, in order to learn more about how to solve the problem, before the development process is actually started.

In some cases, like the Formal Engineering Methods [DSB04], the solution is the other way around: instead of using the existing ones a new methodology is proposed for supporting things like model checking in the entire software development process. An example of this is SOFL [LAKN98] (Structured Object-oriented Formal Language), which includes a specification language, Petri Nets, and data flow diagrams; a complex method combining concepts from formal methods and object oriented development; and proposes a new software process. Another example is MetaFrame [SMC+95], an environment supporting model checking in the entire software development process.

It is not the goal of this thesis to propose a complete solution to the very complex problem of when in the software development process to use formal methods. However, for the proposal we have selected, some recommendations about when it can be used are given below.

VoDKA is developed using an agile software development process. The iterative process has short design cycles and a small set of new features are added to the system in each iteration. After each iteration, a new prototype is produced.

As our proposed approach is almost completely automatic, and provides fast feedback to the users without requiring a lot of effort from them, it can be used in parallel to the different iterations of the development. Once the development team has the very first prototype ready, the tools can start being used. This can continue all the way until the last working version of the system is in place.

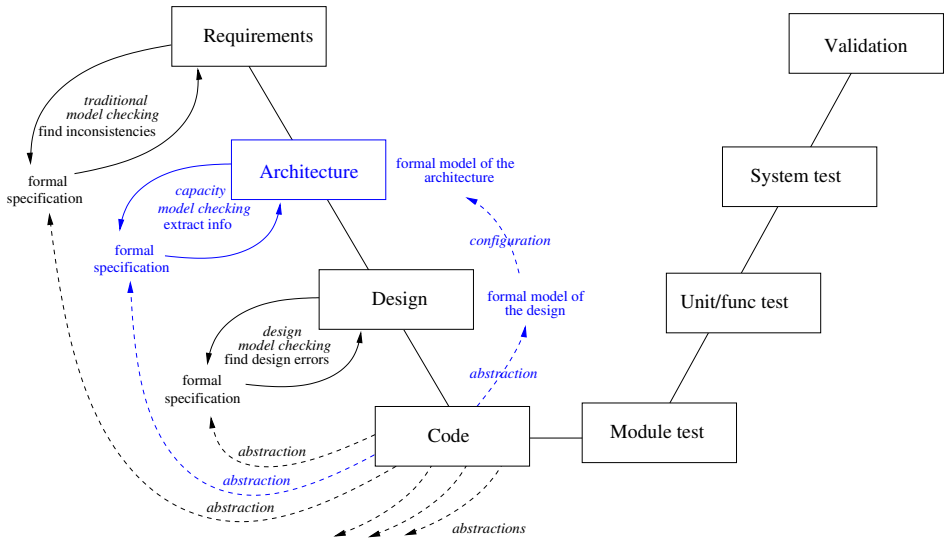


Figure 7.1: Formal methods placed in the V-model schema

For each version, model checking for extracting information of the system can be carried out following our method. The main effort will be to decide how to abstract from the real and detailed version of the code to the *Erlang* model from which we are going to generate the state space. Another continuous effort is to enrich the properties we are going to analyze in the system, in order to extract each time more and better information. It is a natural process to progressively improve the abstraction and the properties in parallel to the development, because each stage the development team is going to know more about the system architecture and what they want to extract from it. It can be seen as an *agile formal verification* of the system, whose complexity increases at the same time that the architecture gets more complex. The results of each application of the model checking do not need to be ready before the next iteration is started; depending on the tame it takes to obtain those results, and how easy is to extract conclusions from them, they will be used in order to improve the design of closer or further away future iterations. The verification process just gives extra information to the development team helping them to improve their decisions.

Another point of view is related to which *level* of the system we are analyzing. Formal methods can be applied to check if system requirements are correct (sound, coherent, consistent, and so on); they can also be applied to the design of the system, looking into the components in order to check if they have errors (liveness properties, bugs in the code, etc.). In our case, we look into something that is between the design of the components and the requirements: the system architecture, which describes how the components interact with each other and how they are configured for each of the deployments.

In Fig. 7.1 we show the different possibilities in the context of the classic V-model. We have extended the figure in order to introduce also the concept of software architecture. The equivalent in the testing side would be testing the software and hardware architecture of the system in different configurations and

deployments.

For using formal methods in different *levels* of a system, we use several kinds of abstractions (both for the specification of the properties and for the model). Normally, abstractions used for checking the requirements are very high level, while the ones used for finding errors in the components of the system, are more detailed ones, therefore more closed to the real source code.

In the case of *VoDKA*, we look into the software architecture. The formal specifications are logical properties that express something related to the capacity of that architecture. At the same time, we go from the real source code of the system to a formal model of the design, and then to the formal model of the architecture. The formal model for the design is the abstracted *Erlang* code, hiding the details that are not relevant for the properties we want to extract. The formal model of the architecture is a combination of the designs of the components with the concrete configuration parameters. We work with a formal model of the architecture still described in *Erlang*, although, as it will be explained in our method in Chapter 9, we sometimes need to translate it into more formal languages in order to take some advantages from them an generate better state spaces.

An example of using similar tools than the ones presented in this thesis, but in a design level and with different goals, is mentioned in Section A.3.

7.4 *Who*: the actors involved

Another question related to formal methods is who should use them. From the different roles that take part in the software development, which of them should be in charge of analyzing the system with model checking? The tester? The designer? The developer? A new specialized role only in charge of formal methods? In some teams strict role descriptions are defined, and in other cases roles are more open. Depending on the size of the team, the same person could also play different roles.

An any case, different formal methods techniques and approaches have different answers to this questions. In some cases regular developers are not familiar with the mathematical background needed in order to apply simulation or theorem proving techniques. In those cases, a more automatic approach, as it is our case with the model checking approach for *VoDKA*, makes easier to introduce the tool inside the development team.

Therefore, if we want to make our method not only valid for *VoDKA* but for any similar *Erlang* system, it seems a good idea to try to hide as much as possible the underlying formal background. Making easy to learn tools, that are effective in finding errors quickly, would help the approach to be adopted.

With model checking, the knowledge needed in order to perform the steps is normally related to the tools used for generating and manipulating the state space, together with the ability for translating requirements into properties in a given logic. In our proposed approach, we have tried to select always the most automatic alternative, and we have developed a GUI for hiding as much as possible from the users.

The goal of this is to get the system architects and the software architects to work with the tool, obtaining feedback that can help them to understand better the performance of their system.

7.5 Our proposed approach

We have tried to propose an approach for using formal methods in an automatic way, applied to an industrial system, integrated as a parallel tool all over the agile development method followed by the design team, and we have tried to extract performance properties.

Our proposal, that will be detailed in the next two chapters, can be summarized as follows:

- *Why?*: we use formal methods because they are a promising way of learning more about the quality of service in a system like *VoDKA*. We want to analyze with formal methods the architecture of the system. The goal is to extract performance information for helping to increase the quality through a better understanding. We try to follow a pragmatic approach with a reasonable time-consumption in the context of the development of the *VoDKA* project.

- *What?*: inside formal methods, we have selected model checking. Simulation, testing and theorem proving could be used as complements to our approach.

- *When? and how?*: we propose to use the method following an *agile formal verification* approach in each iteration in order to enrich the design decisions for the future features that are going to be added to the system. Inside the V-model figure, we work at the architecture level.We use the source code and system configuration as inputs for our approach. The model for the architecture is an abstraction of code and configuration, still done in *Erlang* and then possibly translated. We developed some tools and combined them with others that were already available in order to generate the state space for a given software architecture. We express information we want to extract as logical properties to the model of the system and we model check them against it.

- *Who?*: we propose a method that hides the low level details and automates as much as possible, that could be used by system and software architects without having a very steep learning curve.

In Chapter 8 we describe the tools used and in Chapter 9 we present the method, the experiments and the concrete results we have obtained with this approach.

7.6 *Limits* of the approach

Obviously, apart from the limits of the work carried out in the current and future research, that will be explained in the next part of the thesis, our approach has some intrinsic limitations.

Due to the fact of using model checking, we cannot look into potentially infinite models. We need to abstract from details in order to handle them, what works quite well for the systems we are looking into, because *Erlang* itself hides very well with its *behaviours*. Another limitation of model checking is the size of the state space graph and the time it takes to generate it. We will talk in detail about this in the

next chapters, but for very complex systems, reducing the size of the graph is one of the challenges.

Also, we cannot look into all kind of systems. The method we have developed has as natural target a resources-aware system. If the system is not aware of the underlying hardware limits, we would need to model the hardware and add it as a *context* to the real software model.

Another limitation could be that we cannot hide everything related to formal methods with a GUI. However, this should not be a problem, because simple things can be quite automatically done, and for the more advanced use of the approach the users are going to have some knowledge on writing logical formulas and dealing with model checking.

And finally, the kind of properties we can extract are those for which the information is present in the state space graph. This has some limitations, but as we will see in the next chapters, we have managed to express relevant properties only using the data present in the graphs generated from the model.

7.7 Other approaches for performance evaluation of software architectures

Our approach is very innovative in the tools used and the way of using formal methods for capacity analysis.

In the survey [BMIS04] published by Balsamo et al in 2004, the main approaches for the model-based performance prediction in software development are compared. By model based they mean that the approaches based on analyzing performance by running the system are excluded. The survey is focused in early software performance predictive analysis and discuss how they all integrate in the ordinary software development. They define software performance analysis as the process of first predicting and later evaluating if a system meets its performance requirements.

In the study, the methods for software performance engineering are divided into the ones based on queuing networks (QN), the ones based on extensions of process algebras and Petri-Net-based approaches, the ones oriented more to simulation techniques, and those based on stochastic processes.

The most common ones are those based on QN. Inside them, the ones based on SPE [WS98] (Performance Software Engineering) are quite popular. They use two different models, one for the software execution where the execution behaviour is modelled (an execution graph), and the other for the system execution (a queuing model).

A second kind of approaches inside QN make use of Architectural Patterns. These patterns identify frequently used architectural solutions, and given a way of analyzing performance for those patterns and a way of composing performance analysis, they allow the approaches to extract information from the complete systems. This approach of using components could be seen as similar to our idea of constructing the behavioural state space from the implementation patterns in the *Erlang* source code. But our patterns are directly specified by the developers, while the architectural ones need to be detected at a different phase of the software development.

A third kind of QN-based approaches are based on creating the model from the information present in traces extracted from the dynamic description of the software components. Some of them use labelled transition systems as the way of describing dynamically the system. This could again be seen as similar to the approach we propose for systems like *VoDKA*. However, in our case, LTSs are extracted directly from an abstraction of the source code and are subject of analysis, while in this approach they are created by the designer and used as a way of generating the QN model.

Finally, the forth set of QN approaches includes those that take as input the extensions that had been developed on top of UML for including performance information. From those UML descriptions and diagrams, the model is generated.

In the survey they also claim that *with the increase of software complexity, it was recognized that software performance could not be faced locally at the code level by using optimization techniques, since performance problems often result from early design choices.* It is important to state that although we use the code as input, we do not really evaluate performance at a code level. Instead, we extract the software architecture from the code and then we analyze performance at a higher level. Therefore, we confirm their observation and align with their affirmation.

In the final conclusion, the survey claims that the future of the model-based methodologies for performance analysis and prediction seems to be encompassing the whole software life cycle starting from the very first artifacts, at the software architecture level, and that the automation of that kind of approaches is going to be a key factor. Our approach fits perfectly in this description, and offers a different way of analyzing distributed systems, still aligned with the main messages they propose.

The kind of information that all the described models study is similar to the one used in our approach: platform data, system configuration, capacity of the underlying resources, and what they call the operational profile which characterizes the users of the system and therefore the workload information. However, they are normally designed for helping in the first stages of a cascade-like development, and are less adapted to an agile method focused in working with directly with the source code.

Chapter 8

Tools for state space generation and analysis

Contents

8.1 Introduction

In the previous chapter, the context in which we propose the use of formal verification tools and techniques in order to analyze software architectures, was discussed. In the next chapter, we will introduce a concrete method we propose for extracting system information using formal methods, but before that we describe here the tools are going to use. Some of them have been developed by us in order to solve a need in our method and others are just external tools introduced as part of the steps we propose. In each section, we motivate the development or selection of the tool, and we explain its fundamentals and which parts we are mainly using and how.

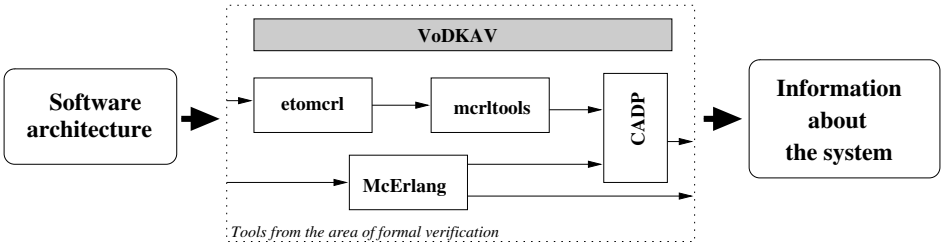


Figure 8.1: Tools for state space generation and analysis

In Fig. 8.1, the context where the tools are used, which will be detailed in Chapter 9, is shown. Basically, the tools, from the area of formal methods, help us in order to go from the software architecture of the system to the automatic extraction of information.

This chapter is structured as follows: in Sect. 8.2 we introduce the tool that has been developed in order to translate *Erlang* source code to μCRL specification. In Sect. 8.3, the μCRL toolset, used for manipulating μCRL specifications and generating their state graph, are described. After that, the CADP tool, utilized to manipulate the state space, mainly performing some reductions by hiding the information not needed and later doing model checking of the properties, is motivated and described. Finally, in Sect. 8.5, we explain an alternative to the use of etomcrl and μCRL toolset: directly generate the state space from the *Erlang* source code using McErlang (*Model checking in/for Erlang*); as McErlang is really a model checker, it can also substitute CADP and directly extract the information about the system.

8.2 etomcrl: translating *Erlang* to μCRL

As explained in Parts I and II of this thesis, the distributed functional language *Erlang* has been originally developed by Ericsson to implement large switching

systems. The *VoDKA* server is just an example, and *Erlang* is nowadays used by several companies mainly for the development of complex distributed control systems. Therefore, in order to be able to apply formal methods to industrial systems, it was interesting to take *Erlang* as the target of our tool.

The language μCRL is a process algebra with data. Several verification tools are available for μCRL and other process algebras, including a tool to create labelled transition systems from specifications. The language and the tool are described with more detail in Sect. 8.3. By having a translation from *Erlang* to μCRL we can apply the verification tools for process algebras and labelled transition systems to industrial code. The translation we carry out is aware of the major design component in the switching software. This knowledge about the software internal implementation patterns is used inside the tool to ensure that the size of the labelled transition system, generated later from the μCRL specification, will be smaller than with a naive translation.

The `etomcrl` tool has been successfully used to transform *Erlang* to μCRL in two industrial case studies, as part of the framework for applying formal methods tools in order to improve the studied software. One of the case studies, the *VoDKA* project, is part of this thesis, and the concrete usage of the `etomcrl` tool is explained in detail in Chapt. 9. The other case study is the control subsystem of the AXD 301 high capacity ATM switch, developed by Ericsson and used to implement, for example, the backbone network in the UK. This study is briefly explained and compared to *VoDKA* in the Appendix A.3.1.2.

8.2.1 Introduction and motivation of the tool

The `etomcrl` tool was originally motivated [ABD04] by the need of verifying a small, but critical, part of Ericsson's AXD 301 [BR98b]. Similar needs were detected while developing this thesis, in order to use verification techniques for analyzing the *VoDKA* project from its source code. Both projects were therefore the case studies that led first the development and later the improvements of the tool.

The industrial case-studies we considered had no up-to-date detailed specification, and could be abstracted to only a few hundred lines of code as a starting point. The real code itself was much bigger but, as we will explain for the case of *VoDKA* in Chapt. 9, we can hide some parts of the system and take only the critical core we want to analyze. The part of the code we are looking into is in both cases written in *Erlang*.

The fact that we are confronted with source code and an out-of-date specification is a rather general phenomena: in some cases verification of software is performed at a rather late stage, when engineers feel that what they have produced is more complex than they understand; in other cases the verification effort starts earlier (we talk about the different approaches in Chapter 7), but the systems are developed by rapid prototyping, and without very complete specifications being available at any time. This is specially frequent when the software developed is innovative and there is not a clear idea of how it is going to be until some experiments are carried out. In both cases the members of the development team want backup by some sophisticated tools that go beyond testing. Tools like model checkers exist, but are not directly applicable to the software they write. Our goal has been to bridge the gap and translate the source code to an input language

for verification, so that a lot of tools are usable for the software. The translation should be effective and rather general, so that the back-end technology can easily be changed for solving different kinds of problems or generating specifications in different languages.

The code of our typical examples is written in *Erlang*, otherwise we could probably have considered using of the comparable initiatives to translate real code into a specification framework, e.g. [CDH00, HP00]. However, the available tools to translate imperative and object-oriented code are too specific to be able to re-write for a functional language, like *Erlang*. Moreover, due to the functional nature of *Erlang* with light-weight concurrency, the target specification language typically differs from a target language for imperative or object oriented languages. Also, in the process algebra attempt one is free to use unbounded data structures, like lists and natural numbers, in the specification. If the actual use of these data structures in the program turns out to have a bounded size, then one is able to generate a finite state space. However, one need not decide on beforehand what the maximal size of these data structures is. For *Erlang* programs it is quite often the case that lists have a fixed maximum length during all possible executions of the program, but that this length differs for different configurations in which the program is used. This fits seamlessly in the process algebra framework. Besides, there is even a stronger extra reason why we developed a new tool, apart from the ones commented above: we are embedding in the translation the knowledge about the implementation patterns used in most of the *Erlang* programs, and we use it in order to generate a simpler specification and, at the end, a smaller state space. That semantics are not embedded in the alternatives for other languages, and this makes our approach different from a theoretical point of view, not only because of using a different programing language.

Erlang is sufficiently different from a process algebra to make it a challenge to come up with an automatic translation. However, the fact of having similar semantics both message-passing distributed functional languages and process algebras makes the selected formalism fit quite well. We choose to pick one specific process algebra with data, viz. μ`CRL` [GR01], since data is crucial in our case. Besides, we had good experiences with the *Open Source* tools [Wou01] that support this language. Another alternative target specification language would have been `LOTOS` [ISO88]; as we have explained, the tool architecture is general enough to make it possible to develop a new back-end.

By our focus from the beginning on two case-studies we ensured that the translation is useful for a rather general class of verification problems. We are able to translate *Erlang* programs that respect the supervision tree and generic server design patterns; hence covering synchronous and asynchronous communication as well as most of the computational aspects of the language.

Process creation is also supported in the translation, if performed by the supervisor design pattern. However, in *Erlang*, a new process can be *spawned* at any time just evaluating a special function. This dynamic process creation at any time is not supported by the μ`CRL` process algebra, which needs all the processes that are going to participate in the model to be defined and started from the beginning in the specification. However, this limitation is not big, because in order to model processes that are spawned in the real program, we can always simulate this by

introducing relatively small changes in the *Erlang* source code. Processes would be started from the beginning: they would be included as part of the structure initiated by the supervision tree, and then their behaviour would be synchronized so that they are not *active* until they receive a special *spawn-like* message.

The verification approach is normally focussed on finding errors in the *Erlang* software and not on proving full correctness of it. That was the case in the AXD case study. With the *VoDKA* server, we use the same kind of tools, but we extended the approach in order to extract information from the state space that sometimes is not a failure but relevant data about the system performance. In both cases, the approach is pragmatic. The traces to failure or counterexamples to properties that we, in the end, obtain by model-checking tools should correspond to traces in the real software. The translation ensures this criteria by its construction.

The description of `etomcrl` is organized as follows: In Sect. 8.2.2, we describe how the difference between *Erlang* and process algebra is bridged. By means of code examples from *Erlang* we show the difficulties we encountered in the translation. The examples presented here are reduced to the most basic concepts. We have included as appendixes some more complete examples with both *Erlang* source code and its translation to μCRL. In Sect. 8.2.3, we discuss the software architecture of the actual implementation of the tool. In Sect. 8.2.4 and Sect. 8.2.5, we present two extensions that we have developed in order to illustrate the flexibility of the tool. Finally, in Chapt. 8.2.6, we talk about some results of using the tool and discuss its potential and limitations in practice.

8.2.2 Bridging the gap between *Erlang* and the μCRL process algebra

8.2.2.1 Processes and communication

Erlang is a language with light-weight processes and asynchronous message passing. The language supports both concurrency and distribution. The concurrent processes run in the same virtual machine (a *node* in *Erlang* terminology) and several virtual machines can be connected to obtain a distributed system. Syntactically there is no difference in communication between processes on the same node and on different nodes; from the programmer point of view, thus, the process distribution is transparent. Of course, distribution gives rise to true non-determinism and a slightly different fault behaviour. An extension to the *Erlang* semantics was proposed recently in order to be able to model that differences [CS05]. For our purpose, it suffices to model both distributed and concurrent *Erlang* processes as truly concurrent processes, like one has in a process algebra. The full non-determinism is covered that way.

Erlang processes communicate asynchronously with each other. Every process has a unique identifier that is used to address in the messages. Every process also has a message queue in which the incoming messages are stored. The virtual machines guarantee that every message is delivered to the queue of the process that the message is sent to. If the receiving process does not (longer) exist, then the message is simply lost without warning. The receiving process actively reads the message buffer by a `receive` statement. This receive statement is blocking as long as the expected message has not arrived.

A straightforward attempt to map *Erlang* processes and communication to a process algebra is to create two process algebra processes: one buffer process and one process to implement the actual logic of the original *Erlang* process. The asynchronous communication is modelled by the synchronizing actions of process algebra. One action pair to synchronize the sender with the buffer of the receiver and one action pair to synchronize the active receive in the process implementing the logic with its buffer.

We have chosen to represent the unique process identifier as data to a general communicating action instead of having unique communicating actions per pair of communicating *Erlang* processes. This way we reduce the amount of actions needed for the translation and make easier later the verification of the state space generated from the specification (where the actions are going to be visible as transitions between states). Having a different action for each couple of processes would complicate the properties to be expressed, removing the possibility of generalizing the formulas.

In *Erlang* programs it is good practice to add your own process identifier to the messages that are sent. In that way, the receiving process is able to respond. The *Erlang* primitive `self()` returns the process identifier and by writing `Pid!{Msg,self()}` one sends a message containing both the term `Msg` and the process identifier of the sending process to the process with identifier `Pid`. With the receive statement one reads a message from the queue. Pattern matching is used to selectively read a certain message from the queue.

For the moment assume that we have embedded *Erlang* communication primitives in the functions `call` and `reply`[1]. The `call` is used to send a message, attached with the process identifier of the sender, and to wait for an answer. The function `reply` is used to return an answer to the caller. This way of embedding low level communication primitives in functions is common practise for industrial *Erlang* code and in the next section we will describe this in more detail.

The following snapshot running in an *Erlang* process describes a simple server that waits in an infinite loop for a client request and replies with an acknowledgement. Variables start with an uppercase character in *Erlang*, constants and functions start with lowercase.

```
loop() ->
  receive
    {request,Client} ->
       reply(Client,acknowledge)
  end, loop().
```

The client to this server evaluates the function `call(Server,request)`, where `Server` is the process identifier of the server and `request` is the message: in this case a constant (called an atom in *Erlang*).

We translate the *Erlang* server process in a μ`CRL` specification with two processes (the buffer and the process implementing the logic), and actions to synchronize the client with this buffer, and to synchronize the process implementing the logic with the buffer.

[1] In *Erlang*, the function embedding is part of the design patterns

The μCRL statement $sum(X:\mathcal{T}, p)$ is shorthand for a non-deterministic choice of all possible values of X of type $\mathcal{T}$ in P.

```
proc server(Self: Pid) =
       sum(Client: Pid,
           receive(Self,request,Client).
           reply(Client,acknowledge,Self).
           server(Self))

proc buffer(Self: Pid, Messages: TermList) =
  b_receive(Self,data(Messages),pid(Messages)).
     buffer(Self,rmhead(Messages)) +
  sum(Msg: Term,
    sum(From: Pid,
        b_call(Self,Msg,From).
        buffer(Self,add(Msg,From,Messages))))
```

The code of the server, for any client *Pid*, receives a request from the buffer and then sends the reply back to the client. The code of the buffer either sends the head of the queue to the server process and removes it from the state (kept as an argument to the process loop) or receives a new message from the client and adds it to the queue.

The process identifier is automatically added by the translation tool as a parameter to processes and communicating actions. The buffer and the process implementing the logic have the same process identifier. This makes easier the translation, because the introduction of the buffers is done in a more transparent way, just by using the right actions in the client, buffer and server processes, and without needing to rewrite the identifiers.

Communication is untyped in *Erlang* and we have to be ready to accept any term in a message queue. Communication in μCRL is specified by pairs of communicating actions; three pairs in our case:

```
receive | b_receive = ex_buffer
call | b_call = in_buffer
reply | replied = sync_reply
```

Therefore, if the action `b_receive` in the buffer synchronizes with the action `receive` in the server process, the message is extracted from the buffer and received by the server. If the action `b_call` in the buffer synchronizes with the action `call` that would be part of the client μCRL specification, the message is stored in the buffer. Finally, if the `reply` action in the server synchronizes with the `replied` action in the client, the answer, that is not modelled through the buffer for reasons we explain later, is sent back.

To simplify reading, we assumed a type *Pid* in the above example, but, in untyped *Erlang*, process identifiers are just terms. In the real translation we follow that concept and use type *Term* instead of *Pid*.

The reader familiar with *Erlang* will have noticed that the FIFO buffer above differs a lot from the semantics of an *Erlang* message queue. In an *Erlang* receive statement one can pattern match on the format of a message. In that way, one can leave certain messages in the queue and selectively take another messages from

it. This is rather difficult to model in μCRL, and one easily ends up in creating a model that causes an infinite state space to be created.

A FIFO queue is insufficient for the client process that communicates with the server above, since processes can freely send a message to the buffer of this client. If another message than the server reply arrives earlier in the message queue of the client, then the client will be blocked forever (actions would never synchronize). The solution to overcome the problem of selective reading of the queue lays in carefully studying what happens in the real *Erlang* code. To our advantage, selectively reading a message from the message queue is only done in very restricted circumstances. Basically the mechanism to read a message other than the first message in the queue is only used for exactly this synchronization. The client adds a special (unique) tag to the message and the server replies with the same tag added. The client is just waiting for any message with the right tag. All other messages that arrive to the queue in-between, are left untouched.

We can model this by having the `reply` action communicate directly to the `replied` action in the client, therewith circumventing the message queue of the client. The differences between the interprocess communication in the *Erlang* source code and the μCRL specification is described in Fig. 8.2.

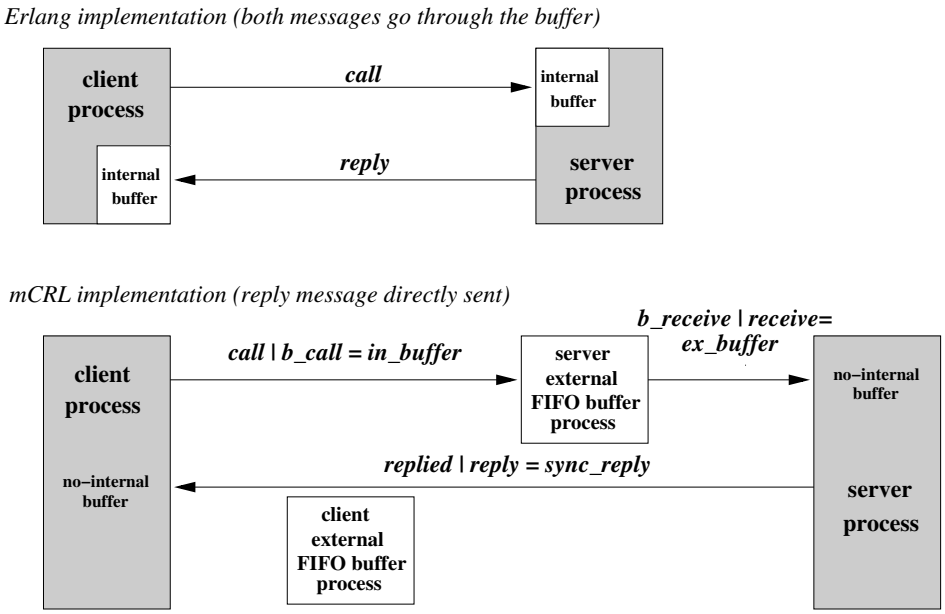


Figure 8.2: Communication in *Erlang* and μCRL for standard processes

This solution only makes sense in a situation where we know which messages are of this special kind and if we know that other messages are dealt with in a FIFO manner. But, that is exactly what we recognized when looking at real industrial projects like *VoDKA* or the AXD 301 switching software. The code is written according to certain design patterns. About eighty percent of the communicating processes implements a server that uses the *generic server* pattern. This server restricts communication in a way that eases the transformation for us[2]. Every generic server has an explicit state defined: the state is passed as a parameter

[2]The restriction in the communication imposed by the design pattern allows a better under-

in the call-back functions. Process state information, like the actual memory on the heap, the content of the message buffer and such is hidden for free. The generic server is an abstraction for debug features that production code typically is attached with. It is an abstraction for the handling of shutdown, code replacement and such. By concentrating on the call-back functions for handling messages, one concentrates on the basic functionality of the server and abstracts from a lot of standard features. In the next section the server pattern and its translation to μCRL are explained in more detail. The supervision tree is another frequently used pattern and is described in Sect. 8.2.2.6.

8.2.2.2 Design pattern: generic server

The *generic server* pattern is used to implement servers in *Erlang*. An abstract version of a server is that of a process which keeps an internal state, waits for an incoming message, computes a response message depending on the incoming message and state, and replies to the message and updates the state. The `gen_server` module implements the generic parts of the server while the call-back module implements the specific functionality (the logic) of a particular instance of the server, i.e. the computation of the response message and the new state.

The above description is, on purpose, an oversimplification of the generic server behaviour. The behaviour also takes care of a uniform way of error handling, of a uniform debugging facility, of monitoring nodes and observing whether clients are still alive, etc. In that way, the programmer really only needs to concentrate on the logic of the server. That, on its turn, allows us to easily abstract from a lot of details that the code would have if not implemented in the generic way. Our translation tool can, by means of this generic behaviour, concentrate on the logic and abstract from the implementation details of error handling, debugging, etc.

The simplified version of the generic server is just a small extension of the server given in Sect. 8.2.2.1. We add state as a parameter of the loop and whenever a messages arrives, we need to call a function to evaluate a reply and to update the state. The generic server distinguishes three kinds of messages: *call*, *cast* and *info*. A call is a synchronize event (built on top of the asynchronous interprocess communication we have explained), where the client waits for a reply. The cast is the asynchronous version of the call, and the info messages serve the special purpose to deal with error events and such. In this description we only consider the *call* messages, but the actual translation handles the full generic server with all three kinds of messages.

```
loop(M,State) ->
    receive
       {call,Msg,Client} ->
          {reply,Reply,NewState} =
            M:handle_call(Msg,Client,State),
          reply(Client,Reply),
    end, loop(M,NewState).
```

standing of complex systems. The same restriction that makes the system easier to understand for the engineers is making it easier for us to translate the system to the clean framework of process algebra.

The variable `M` contains the name of the module in which the function `handle_call` is implemented. This is the so called *call-back module*. The server loops continuously keeping the state of the process as an argument. Whenever a `call` message is received, the function `handle_call` in the call-back module is evaluated in order to compute the reply and the new state. Then the reply is sent back to the client and the computed state passed recursively to the loop.

Remark that the programmer uses the standard generic server component and only provides the call-back module when implementing a server. The generic part is static and stable over the years. Therefore, we can take the semantics of the generic part for granted and use it in our translation.

A typical example of a call-back module is given below. It implements a very simple server process part of a *VoD* system that may receive either a *lookup* message or a *play* message. The state of the server is a record which includes the list of available *Media Objects*, the current connections (movies that are already being streamed), the bandwidth currently used by the connections, and the maximum number of connections and maximum bandwidth that are allowed.

```
handle_call({lookup, MO, Profile}, From, State) ->
    case (check(MO, Profile, State)) of
        true ->
          {reply, {lookupAns, ok, MO, Profile}, State};
        false ->
          {reply, {lookupAns, fail, MO, Profile}, State}
    end;

handle_call({play, MO, Profile, Dest}, From, State) ->
    {reply, ok, update_state(MO, Profile, State)}.
```

Whenever a client sends a lookup message, it includes the identifier of the *Media Object* and a profile with the bandwidth required. With that information, a function *check* can determine if there are enough resources for sending the media to the client, and answer according to this. Whenever a *play* message is received, the state is updated and an acknowledge message is sent back to the client. Internally, the actual stream of the movie to the process `Dest` would be created but that low level part is not needed for this explanation.

The *Erlang* functions `check` and `update_state` should also be implemented in this call-back module, but its actual implementation is not relevant here. The behaviour also provides functional embeddings of the communication primitives, similar to what was used in Sect. 8.2.2.1. A client would evaluate the `call` function. The implementation takes care of adding a unique tag and the process identifier of the client to the message. It also takes care of waiting for the arrival of a reply from the server with exactly the same unique tag in order to proceed.

Thus, a typical client that would do a lookup and a play is implemented in *Erlang* by:

```
client(Server, MO, Profile, Dest) ->
  Nr = gen_server:call(Server,{lookup, MO, Profile}),
  gen_server:call(Server,{play, MO, Profile, Dest}).
```

The translation of both server and client is similar to the translation given in Sect. 8.2.2.1. Fig. 8.3 updates Fig. 8.2 with the synchronization actions used for

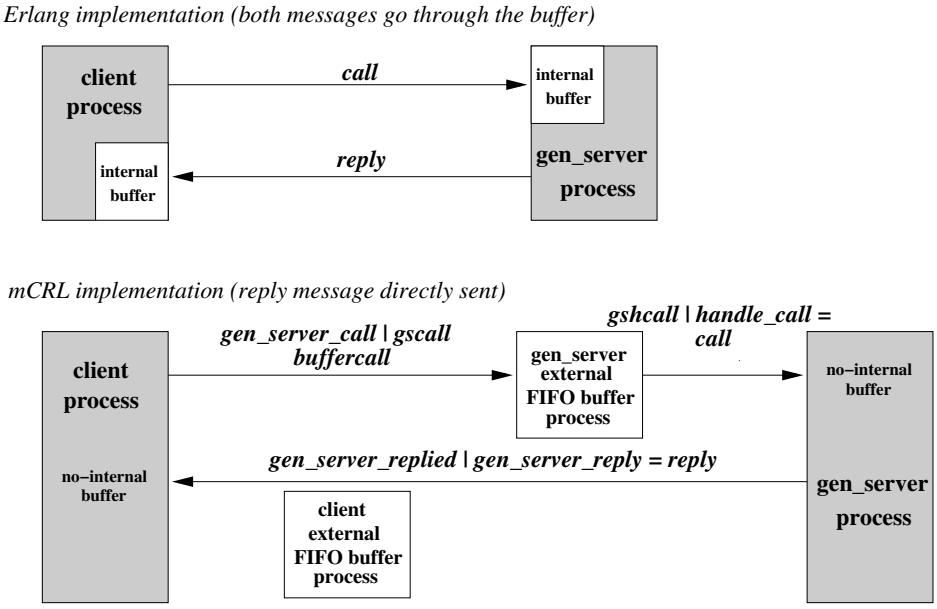


Figure 8.3: Communication in *Erlang* and μCRL for generic servers

generic servers (the main changes are the names of the actions that are used; the essence of the translation remains the same). For example, now instead of the receive action, we use a `handle_call` action and a non-deterministic choice is used to be able to either receive the lookup or the play message. The buffer is basically the same, apart from the changed names of the actions. The three pairs of actions we use for the generic server are:

```
gen_server_call | gscall = buffercall
handle_call | gshcall = call
gen_server_reply | gen_server_replied = reply
```

One option for translating the generic server and the call-back module would be to perform the complete translation of both modules. But as we have already explained, the goal of the `etomcrl` tool was to embed the semantics of the generic server behaviour in order to simplify the translation steps, the generated specification and therefore the state graph. Thus, in order to carry out the translation we have to take into account the knowledge we have about the generic server implementation: the `handle_call` function is translated in a non-deterministic choice between the alternatives and embedded in a server loop. The last argument of the `handle_call` needs to be converted in the last argument of that loop, and the rest of the arguments are left in order to pattern match with the action on the other side of the inter process communication. We leave the explanation of how this is handled in detail when complex pattern matches are involved for the next sections.

We provide the translated specification of the server process in μCRL:

```
proc server(Self:Term,State:Term) =
   sum(From:Term,
     sum(MO:Term,
       sum(Profile: Term,
```

```
          handle_call(Self, tuple(lookup,
                      tuple(MO, tuplenil(Profile))),From).
          (gen_server_reply(From,
              tuple(lookupAns,
                 tuple(ok,tuple(MO,tuplenil(Profile)))),
              Self).
          server(Self,State)
          <| eq(equal(check(MO,Profile,State),true),true) |>
          (gen_server_reply(From,
              tuple(lookupAns,tuple(fail,tuple(MO,
                            tuplenil(Profile)))),Self).
            server(Self,State)))))) +
  sum(From: Term,
    sum(MO: Term,
      sum(Profile:Term,
        sum(Dest:Term,
          handle_call(Self,
            tuple(play,tuple(MO,
                        tuple(Profile,tuplenil(Dest)))),From).
            gen_server_reply(From,ok,Self).
            server(Self,update_state(MO,Profile,State))))))).
```

The somewhat obscure notation for the if-then-else statement in μCRL is $then \triangleleft if \triangleright else$. Tuples are inductively defined using the constructors `tuple` and `tuplenil`.

The μCRL functions for `check` and `update_state` would be almost equal to the *Erlang* counterparts. In Sect. 8.2.2.8 the translation of such purely computational functions is explained in more detail.

The specification for the client in μCRL would be as follows:

```
proc client(Self:Term,MO:Term,Profile:Term,Dest:Term,Server:Term) =
  gen_server_call(Server,tuple(lookup,
                               tuple(MO,tuplenil(Profile))),Self).
  sum(Nr:Term,
     gen_server_replied(Self,Nr,Server)).
  gen_server_call(Server,tuple(play,tuple(MO,
                             tuple(Profile, tuplenil(Dest)))),Self).
  sum(Free:Term,
     gen_server_replied(Self,Free,Server)).
```

The last *sum* in the client is generated automatically, because the `call` function is always returning a result. In *Erlang* one may choose to ignore the result, but in μCRL we have to explicitly bind it to a variable (`Free` in this case). This already indicates a subtle difference between a return value of a function in *Erlang* and a communication action in μCRL. This issue is explained in more detail in the next section as it is not specific for the generic server, but a more general phenomena.

Translating the different *handle_call* clauses into non-deterministic choices is not the only alternative and, in fact, it has some limitations. In *Erlang*, if the first clause of a function pattern matches with the arguments, the rest of the clauses are not going to be taken into account (they are ignored). However, in the μCRL translation we have just proposed, all the possible matching clauses would be selected and therefore would potentially synchronize. This introduces different

semantics in the translation, originating non-desired paths in the resulting state space. Due to the clean way of programming used in the industrial code we have translated, and also to the kind of properties we were looking to, the difference in the semantics has not been a real problem neither a limitation for our experiments.

Nevertheless, an alternative would be a translation using an if-then-else approach similar to the one that is introduced in the first part of Sect. 8.2.2.4 for the pattern matching in function clauses. Still there would be some differences in the way some of the arguments are translated. In this approach, the conditions of the 'if' are extracted from the pattern matching, and destructors are used if needed inside the translation of the clauses. This alternative is illustrated with an example in the second part of Sect. 8.2.2.4.

Currently the `etomcrl` tool uses the non-deterministic approach, and a new version with the if-then-else alternative is left for future work.

In this section we have shown the basic principle of translating generic servers into μCRL. We left out the buffer, since that was presented in the previous section. Our tool handles real servers, which are more complicated than the example shown here, but the basic ideas are captured in this section. Our example call-back module is rather simplistic and in reality there are some issues that complicate matters. In the next few sections we focus on those complications.

8.2.2.3 Functions with side-effect

As we have explained in Sect. 3.3, significant difference between *Erlang* and a process algebra is that the latter forces to separate computation from communication. Computation is only accepted in the data part of the process algebra with data, and can therefore only take place in the arguments of the actions, and never at the same level as the actions themselves. In *Erlang*, in contrast, a function that performs some calculations can also communicate. Thus, such a function has communication as side-effect of the computation. Of course, the function can call other functions that have side-effects and as such we can get deeply nested integration of computation and communication. Here, our first task is to identify *Erlang* functions with side-effect from the pure computations. These two classes of functions are translated differently.

Supported by industrial experience, we only consider *Erlang* programs that respect the behaviours like the generic server; that is the case of *VoDKA*, were the control subsystem is heavily based on that kind of implementation patterns. Functions are classified as functions with side-effect when they make use of a communication function (like `call` or `reply`). By analyzing the call graph of all involved modules, we can syntactically split the *Erlang* functions in the two demanded categories.

As seen in the previous examples, the generic server primitives for communication in *Erlang* are translated to actions in μCRL. In addition, `etomcrl` supports user defined actions which are also considered as side effects and translated to μCRL actions. User defined actions can be used for verification purposes: they can be utilized in order to reduce the complexity of the logical properties expressed over the state space generated from the specification. One example of user defined actions usage would be to give visibility to a critical section in a concurrent/distributed algorithm, including two extra actions, one taking place when the evaluation thread

enters the critical section and one when it goes out of it. Other function with side effects in *Erlang* are the functions for using the standard output, but calls to those functions can be ignored in the translation.

In the remainder of this section we focus on the part with side-effects, and how the computation and communication are separated in these functions. Issues related to the purely computational part are discussed in Sect. 8.2.2.8.

The problem with nested side-effects is best illustrated by an example. Assume an *Erlang* process that calls a function `p` in order to communicate its result. The function `p` itself contains a side-effect:

```
loop(X) -> s(p(X)),
           loop(X).

p(X) -> Y = f(X),
        s(g(Y)),
        h(Y).
```

where `f`, `g` and `h` are pure computations and `s` is one of the side-effects, e.g., the `reply` function.

`s` needs to be translated into an action in μCRL, but what should be done with `p`? It cannot be an action because it involves computation and even because nested actions are not allowed in μCRL, but it can neither be just computation because it has side-effects inside. We therefore need to translate it to:

```
proc loop(X:Term) =
     s(g(f(X))). s(h(f(X))). loop(X)
```

We obtain this translation by a recursive source code transformation of the *Erlang* functions. In this transformation we lift all side-effect functions to the highest level and push pure computations down by duplicating them. Thus, we translate on the source code level all functions with side-effect to functions that look like:

```
p(X) -> s1(---),...,sN(---),---.
```

where `---` stands for pure computations and `si` are functions with side-effects.

We use standard techniques of inlining computations and the less standard technique of composing the destructor functions that enable a pattern match.

The variables can be used separately for further computation. In μCRL these variables have to be replaced by destructors of the data structure (we will see this with more detail when analyzing the pattern matching, in the current example the destructor is not needed). Whenever we encounter a statement `Y=p(X)` in the code, we could bind the variable `Y` to the last pure computation of the function `p` and substitute this in the μCRL code. Thus, we could inline the side-effect functions as actions in place of the call to `p`. However, the attentive reader has probably already noticed that this cannot work for recursive functions with side-effects (in our example, if `p` had been recursive); without knowing the number of recursive iterations, one is unable to unfold the definition and hence unable to inline the exact number of side-effects. Neither does this work for functions that perform a side-effect and cannot be de-composed. As an example, consider the simple *Erlang* program that performs a side-effect on every element of a list and returns `ack` as

a result. All results are stored as a parameter of the function, i.e., the list of `ack` returned values is increasing.

```
loop(Xs,Rs) -> R = p(Xs), loop(Xs,[R|Rs]).

p(Xs) ->
    if
(Xs==[]) ->
    ack;
true ->
    s(head(Xs)), p(tail(Xs))
    end.
```

The standard solution to deal with recursive functions when writing a compiler is to implement a stack data structure to store the return values of the recursive calls. We adopt this idea, where the stack is implemented as a μCRL process (already explained as example in Section 3.3) and push and pop operations are communicating actions. With a source-to-source transformation we make sure that all functions with side-effects are in the previously mentioned format with either a pure computation as last expression or a call to another function with side-effect. We replace all pure computations by a push on a stack and pop this value in the code where we call the function.

The above *Erlang* example is translated to the μCRL code below (where the stack itself is omitted):

```
proc loop(X:Term, Rs:Term) =
       p(X).
       sum(R:Term,
           pop(R).loop(X,cons(R,Rs)))

     p(X: Term) =
       push(ack)
       <| eq(X,nil) |>
       (s(head(X)).p(tail(X)))
```

By using the stack and pushing pure computation inside side-effect functions, we can deal with nested side-effects. The stack alternative also provides a solution for the case where in the above match `Y=p(X)` the variable `Y` is replaced by a complicated pattern with several variables. *Erlang* supports a kind of pattern matching in which one can match the result of a function to a pattern of a data structure. Variables are bound to parts of the data structure. The only thing we have to add to our translation is a *sum* construct for every occurring variable in the pattern.

An example of this more complex pattern would be the next *handle_call* clause inside the call-back module of a generic server. *send_all* is a function with side-effects that sends a message to a group of processes and gathers the answers in a list. The list is pattern matched in order to take only the first option (trivial algorithm just useful for the example) and then a message with the cost of that option is sent as answer.

```
...
handle_call({lookup,MO,Profile}, From, State) ->
    [Head|Tail] = send_all({lookup,MO,Profile},get_storages(State)),
    {reply, {lookupAns,cost([Head],State)}, State};
...
```

The translation to μCRL follows all the rules we have explained, and introduces the *sum* construct for `Head` and `Tail` in the pattern matching, calling after that to the action `pop`, that will synchronize with the complementary `push` function inside the translation of `send_all`.

```
serverloop(MCRLSelf:Term,State:Term) =
sum(From: Term,
  sum(MO: Term,
     sum(Profile: Term,
           handle_call(MCRLSelf,tuple(lookup,
                       tuple(MO, tuplenil(Profile))),From).
           send_all(MCRLSelf,tuple(lookup,
                                   tuple(MO, tuplenil(Profile))),
                                   get_storages(State)).
           sum(Head: Term,
             sum(Tail: Term,
                 pop(cons(Head,Tail)).
                 gen_server_reply(From,
                           tuple(lookupAns,
                                 tuplenil(cost(cons(Head,nil),State))),
                           MCRLSelf).
                 serverloop(MCRLSelf,State)))))) +
...
```

In the communicating action we can simply repeat the pattern, since μCRL supports that kind of pattern matching as well. Note that the introduction of the *sum* construct is only used for the matches of patterns with functions that contain side-effects. A match with a pure function is translated differently, as explained in the next section.

Although the stack process solves our problem of translating nested side-effects, we also have to pay the price of more communication in the model and therefore an increased state space of the system. Moreover, the duplication of the pure functions gives rise to longer computation times in the model than in the real implementation. Therefore, whenever it is possible, the first solution of inlining the side-effects is used.

8.2.2.4 Pattern matching in the communication part

As we have explained, the problems and solutions related to the translation are different if the pattern matching appears in the part of the *Erlang* program with side-effects or in the side-effect-free part. In the first case, the code is going to be translated to processes that communicate using actions that synchronize. In the second case, the source code is going to be translated into μCRL rewriting rules. Here we only consider the first case, the second case is described in Sect. 8.2.2.8.

In the previous section we have shown how one particular kind of pattern matching, when the result value of a function with side effects is pattern matched, is elegantly translated by using communication via a stack process. In this section we focus on two pattern matching possibilities that differ enough in both languages to make the translation non trivial: pattern matching in function clauses and in communication primitives.

- Pattern matching in function clauses:

  Process definitions in μCRL can only have variables as parameters, c.f. the definition of client and server in Sect. 8.2.2.2; and there is only one clause per process. *Erlang* functions that are translated to μCRL processes may have several clauses in which pattern matching decides which clause is evaluated. As an extra complication, the *Erlang* function might be defined in such a way that for a certain input there is no matching clause, which makes the process evaluating the function crash. This crash should show in the translated μCRL model as well, since it is valuable information in the verification process.

  Several *Erlang* function clauses can easily be combined in only one `case`-statement, but that does not solve the problem. The pattern matching in the `case` is equivalent to pattern matching on function clause level. We treat those therefore similarly.

  First, we compute the discriminating pattern to select a certain clause (c.f. [Wad87]) and we use a nested if-then-else structure to determine which part of the function to evaluate. This if-then-else can later be directly mapped to μCRL.

  In that translation to μCRL we add a last clause in which the process deadlocks if none of the alternatives matches.

  Second, we replace the patterns in the arguments of the function to variables and replace bindings caused by these patterns to destructor functions. As an example, consider the following *Erlang* function, where `s` is again an *Erlang* communication primitive:

```
loop(X,[]) -> s(done), loop(X,X);
loop(X,[Head|Tail]) -> s(Head),loop(X,Tail).
```

  The elimination of the pattern matching in this code requires two destructors, viz. `hd` and `tl` to extract the head and tail of a list. With those two destructors the code is transformed to the following *Erlang* code:

```
loop(X,Arg1) ->
   if
     nil == Arg1 ->
       s(done), loop(X,X);
     is_list(Arg1) ->
       s(hd(Arg1)), loop(X,tl(Arg1))
   end.
```

The fact that *Erlang* is a dynamically typed language enables the code to fail for calls where `Arg1` is not a list, therefore the last condition in the `if`-statement above is not just *true*. In the translation to μ`CRL` a third alternative is added, reflecting this possible type error in *Erlang*.

```
proc loop(X:Term, Arg1: Term) =
      s(done).loop(X,X)
      <| equal(nil,Arg1) |>
      (s(hd(Arg1)).loop(X,tl(Arg1))
       <| is_list(Arg1) |>
       delta)
```

In general the conditions to check are more complicated than only checking whether an argument is a list or the empty list. We need to bind variables to terms in order to use them in the expressions and sometimes we even need destructor functions in the conditions, for example if we want to check whether the head of a list is equal to the integer one[3]. However, there are only finitely many possible patterns in *Erlang*. The simplified version of the computation of the conditions for given pattern P and expression E, where only lists, integers, and variables are considered is given below. The function returns a condition and a set of variable bindings[4].

$$cond(P,E) = \begin{cases} \langle true, \{P \mapsto E\}\rangle & var(P) \\ \langle is\_list(E) \wedge \phi \wedge \psi, \sigma \cup \tau\rangle & P = [H|T] \\ & \langle \phi, \sigma\rangle = cond(H, hd(E)) \\ & \langle \psi, \tau\rangle = cond(T, tl(E)) \\ \langle equal(P,E), \emptyset\rangle & \text{otherwise} \end{cases}$$

A more complicated version of the above function is successfully used in our source-to-source transformation to map different patterns in function clauses to variables in the arguments of the clauses and nested conditions in the body of the clause.

- Communication primitives:

  The above described function clauses are translated into μ`CRL` process definitions. For function clauses that are communication primitives and that are translated to μ`CRL` communicating actions, a similar pattern matching transformation is necessary. In this case, however, one cannot introduce the if-then-else construct in the same way.

  As an example, consider an extremely simple `handle_call` inside a *VoD* system generic server process. The client can play the movie, stop it or ask for the status (internal streaming is hidden in this pseudo-code). T

[3]The if-statements in *Erlang* do not allow destructors in the conditions, therefore, we use nested case-statements instead of the if-statement, but explaining it by means of an if-statement is clearer.

[4]The set of variable bindings is a list in the real implementation, where variables that have been bound before need to be matched against a value if they occur more than once in the pattern.

```
handle_call({play,mo},Client,{mo,Status}) ->
  {reply,ok,{mo,play}};
handle_call({stop,mo},Client,{mo,Status}) ->
  {reply,ok,{mo,stop}};
handle_call({status,MO},Client,{MO,Status}) ->
  {reply,Status,{MO,Status}};
```

The `handle_call` function is translated in a non-deterministic choice between the alternatives and embedded in a server loop, as explained in Sect. 8.2.2.2. Here we have two matches that need to be translated differently. The generic server loop has `State` as a parameter and the tuple {`MO,Status`} should be decomposed as described in the previous section. The message (and similarly the client identifier) should be treated differently. For those parameters, the variables are isolated and put in a *sum* construct. The matching is done by the pattern matching mechanism of μCRL.

```
server(Self:Term,State:Term) =
 sum(Client: Term,
  handle_call(Self,tuple(play,mo),Client).
  reply(Client,ok,Self).
  server(Self,tuple(mo,play)))
 +
 sum(Client: Term,
  handle_call(Self,tuple(stop,mo),Client).
  reply(Client,ok,Self).
  server(Self,tuple(mo,stop)))
 +
 sum(MO: Term,
  sum(Client: Term,
   handle_call(Self,tuple(status,MO),Client).
   reply(Client,element(2,State),Self).
   server(Self,tuple(mo,element(2,State)))))
```

Of course, we use the knowledge we have on our communication primitives to decide which parameters need to be transformed to match on the process level and which are to be transformed in a *sum* construct. Typically the matching on the process level is translated source-to-source, whereas the introduction of non-determinism and *sum* construct is left to a later stage.

Lacking in the above translation is the introduction of the conditions that we compute for the pattern match. A programmer could easily handle the same message in two different clauses of the `handle_call` function by differentiating the state in which the message arrives. This way of programming is de-recommended in the style guides, but occurs now and then in code fragments. The solution that is currently implemented in the `etomcrl` tool is to introduce assertions that are only going to be true when the arguments match. This can be better illustrated with a modification of the previous example:

```
handle_call({play,mo},Client,{mo,play}) ->
  {reply,error,{mo,play}};
```

```
handle_call({play,mo},Client,{mo,Status}) ->
  {reply,ok,{mo,play}};
handle_call({stop,mo},Client,{mo,Status}) ->
  {reply,ok,{mo,stop}};
handle_call({status,MO},Client,{MO,Status}) ->
  {reply,Status,{MO,Status}};
```

In this case, the translation would be the following one:

```
server(Self:Term,State:Term) =
 sum(Client: Term,
  handle_call(Self,tuple(play,mo),Client).
  assertion(equal(element(int(s(s(0))),State),play)).
  reply(Client,error,Self).
  server(Self,tuple(mo,play)))
 +
 sum(Client: Term,
  handle_call(Self,tuple(play,mo),Client).
  reply(Client,ok,Self).
  server(Self,tuple(mo,play)))
 +
 sum(Client: Term,
  handle_call(Self,tuple(stop,mo),Client).
  reply(Client,ok,Self).
  server(Self,tuple(mo,stop)))
 +
 sum(MO: Term,
  sum(Client: Term,
   handle_call(Self,tuple(status,MO),Client).
   reply(Client,element(2,State),Self).
   server(Self,tuple(mo,element(2,State)))))
```

Where an assertion is introduced after the first *handle_call* in order to ensure that it is only true if the second element of the `State` is the atom `play`.

This solution works for most of the properties and examples we have studied, but introduces semantical differences with the original *Erlang* implementation. If the properties are not written carefully, taking into account the assertions that are false, we could extract at some point wrong information from the state space.

An alternative that would improve the translation would be to put conditions in the loop that correspond with the possible patterns of the state and only then non-deterministically match the possible messages. With this alternative solution, that is left for future work in the `etomcrl` tool implementation, the generated μ`CRL` specification would be:

```
server(Self:Term,State:Term) =
 sum(Client: Term,
 (handle_call(Self,tuple(play,mo),Client).
  reply(Client,ok,Self).
  server(Self,tuple(mo,play)))
```

```
  <|equal(element(int(s(s(0))),State),play)|>
    handle_call(Self,tuple(play,mo),Client).
    reply(Client,error,Self).
    server(Self,tuple(mo,play))))
 +
 sum(Client: Term,
  handle_call(Self,tuple(stop,mo),Client).
  reply(Client,ok,Self).
  server(Self,tuple(mo,stop)))
 +
 sum(MO: Term,
  sum(Client: Term,
   handle_call(Self,tuple(status,MO),Client).
   reply(Client,element(2,State),Self).
   server(Self,tuple(mo,element(2,State)))))
```

The code above is less readable this way but the semantics are much closer to the original *Erlang* program.

In all the previous examples we have used the non-deterministic approach for translating the *handle_call* clauses. For the original example, the translation with an if-then-else approach would be a bit different:

```
server(Self:Term,State:Term) =
 sum(Client: Term,
  sum(Message: Term,
   handle_call(Self,Message,Client).

   (reply(Client,ok,Self).
   server(Self,tuple(mo,play))
   <|and(
     equal(element(s(0),Message),play),
     equal(element(s(s(0)),Message),mo))|>

     (reply(Client,ok,Self).
     server(Self,tuple(mo,stop))
     <|and(
       equal(element(s(0),Message),stop),
       equal(element(s(s(0)),Message),mo))|>

       (reply(Client,element(2,State),Self).
       server(Self,tuple(mo,element(2,State)))
           <|and(
             equal(element(s(0),Message),status),
             equal(size(Message),s(s(0))))|>
     delta)))
```

The code is now less readable for a programmer adapted to *Erlang*, but the semantics are conserved after the translation. In this case the issue is that the first `handle_call` is going to synchronize always, and that has to be taken into account when writing the properties about the generated state space. But the advantage is that no new paths are generated in the state space

created from the specification (in the non-deterministic approach this could be the case).

8.2.2.5 Pattern matching a pure function return value

In *Erlang*, functions return a value that can be pattern matched. We have seen this already in Sect. 8.2.2.3, where the pattern was a simple variable. If we match such a pattern with an expression that contains a side-effect, we have shown a translation using a stack implementation. However, if the expression is a pure function, we use the pattern matching function described in this section. We substitute all occurrences of the variables in the match by their corresponding value.

If the expression evaluates to a value that does not match the pattern, the *Erlang* process evaluating the pattern matching will crash. Thus, if the result of the evaluation is an empty list, whereas it is matched with a non-empty list, then the process in which this match is evaluated will crash. We want to be able to observe those crashes, but need not necessarily translate the crash to a deadlock of the process, like we do in case of the function arguments that do not match. Instead we introduce a new action *assertion*, which is false if the conditions are not fulfilled, i.e. when the *Erlang* process would have crashed. This assertion is easily merged in the sequentialized form in which all functions with side-effect are written.

As an example of this, we can consider the following *Erlang* code inside a side-effect function (for example inside a handle_call):

```
[{Bandwidth, Cost, ProcId}] =
       scheduling(MO, Profile, Movies,
                  UsedBandwidth, Connections,
                  MaxConnections, MaxBandwidth,
                  Cost, self())
```

Its translation would introduce an assertion saying that the returned value should be a list, and that the head of that list should be a tuple of size three.

8.2.2.6 Design pattern: supervision tree

As we have explained in Sect. 8.2.2.2, *Erlang* behaviours are reusable software components that implement the generic part of different kinds of processes. The programmer using them, only needs to provide the call-back module specifying the desired behaviour for the generic implementation. The supervision tree is one of the main generic servers provided with the *Erlang* development platform.

One of the key features of most distributed systems, in particular those for which *Erlang* is used, is fault tolerance. *Erlang* supports fault-tolerance by means of the supervision tree, a structure where the processes in the internal nodes (supervisors) monitor the processes in the external nodes (workers).

Systems should be able to continue working even when some of the modules of the software crash. In order to provide this feature, the programmer would have to implement a lot of extra code to take care of all the possible failure situations and provide some kind of recovering mechanisms. A big part of this code and mechanisms would be repeated in every distributed system; in *Erlang*, all this common

infrastructure has been extracted to a generic supervision tree design pattern that can be parameterized by the user in order to obtain the desired behaviour.

In order to provide fault tolerance features, a supervision tree is build on top of every *Erlang* process. The processes in the internal nodes of the tree are called supervisors, and they are in charge of monitoring any possible crash of their children; and the processes in the leaves are called workers, and have the actual implementation of the application logic. A system is started by creating the top node of a supervision tree (instantiated with a call-back module). Every supervisor node creates all its children and monitors them.

The creation of the processes architecture of the system is encoded inside the supervision tree initialization. This fact can be used in order to extract the processes of the system from the source code and the input (configuration) provided by the user. Process algebras allow the creation of new processes, but the set of tools developed for μCRL does not support this feature. We partially evaluate the supervision tree, using the fact that we know the semantics of that design pattern, in order to obtain its structure and a list of all created worker processes.

The use of the supervision design pattern is so common that using it to find the created processes is no severe limitation. We cannot handle *Erlang* applications in which processes are spawned outside the scope of the supervision tree, but these are not commonly encountered in production code.

In this thesis we do not look into the fault tolerance of the system. In that sense, it is enough if in the `etomcrl` tool we only use the supervision structure in order to know the processes that are going to be started and generate the specification for them. Thus, in the model we do not need to restart a crashed process. However, we use the translation for verification and system analysis purposes and we are often content with finding a place where a process can crash. In the cases in which we are interested in errors that occur after recovery, we will have to use an special approach where some of the fault-tolerance related information is kept in the model [BFD05].

8.2.2.7 Higher-order functions

Erlang is a functional language that supports higher-order functions, something which most specification languages avoid for the inherent complexity of the analysis. The expressiveness of a higher-order function is as useful for a good program as design patterns. Therefore, it is a pity that μCRL is a first order language.

Since higher-order functions are a real extension to a language, there is no simple way of translating these functions to first-order variants. Luckily, most of the *Erlang* code on our case-studies only uses a few predefined higher-order functions, like `map`. We therefore designed the translation to handle only those special cases that we encountered, like we only handle a few design patterns. We defined a source-to-source transformation on the selected functions to flatten them to first-order alternatives. Any occurrence of the function `map`

```
map(fun(P) -> f(P,E1,...,En) end, Xs)
```

where `P` is a pattern, `Xs` an expression returning a list and `E1`,...,`En` arbitrary expressions, is replaced by a call to a unique function
`map_f(Xs,E1,...,En)`. The unique function is added to the code and defined as:

```
map_f([],Y1,...,Yn)     -> [];
map_f([X|Xs],Y1,...,Yn) ->
  [f(X,Y1,...,Yn)| map_f(Xs,Y1,...,Yn)].
```

Yi are the variables of the expressions Ei. For the following simple *Erlang* example that uses the function `map` of the lists module:

```
...
lists:map(fun(P) ->
             update_option(P, {1,State}, true) end, Options)
...

update_option(Option, Info, Flag) ->
    do_some_update(Option, Info, Flag).
```

The translation would look like the following one, where the first line is the new way of calling to the generated function. `State` is the variable extracted from the expressions in this case.

```
...
map_update_option(Options, State)
...

map_update_option(nil,State) = nil
map_update_option(cons(P,Ps),State) =
            cons(update_option(P,
                               tuple(int(s(0)),tuplenil(State)),
                               true),
            map_update_option(Ps,State))
```

Although for many functions a similar transformation pattern can be used, there is no general way of translating higher-order concepts into μCRL.

8.2.2.8 Data and pure functions

To translate *Erlang* to μCRL, the data representation in *Erlang* needs to be transformed to the data representation in μCRL. Although *Erlang* has a fixed and small set of constructors, the translation of the data part is more complicated than one would wish. Basically it is a syntactic conversion of constructors, destructors and selectors. The latter two implemented as μCRL functions that directly correspond to the *Erlang* functions. However, an obstacle in this is that not all *Erlang* data structures are inductively defined. The integers, which most programming languages support, are probably the best example of that. In μCRL all data structures need to be defined inductively and the advised way of defining integers is by means of naturals, which are represented as zero and its successors. This might be a theoretically rather clean approach, but in practise it means unreadable specification for larger numbers, slow computations and tools that complain about a too deep term depth when numbers get large.

Another obstacle is that syntactic equality is not a predefined relation, but that this relation has to be specified. In particular for rich sets of data structures (which we use), this results in a large amount of defining rules.

In μCRL the data part and communication part are strictly separated. Data is given in *sorts*, which describe the data type and the operations on it. These operations are strongly typed, using the names of the defined sorts. For example, a sort `Natural` is defined by a constructor 0 and a successor function s to build numbers of arbitrary size.

Typical for process algebras is that such primitive types are not built-in, but are constructed in this rather involved notation. Abbreviations for a finite subset of the natural numbers can easily be specified as well.

The *Erlang* notion of a function is called a `map` in μCRL, and is specified by the type and a set of rewrite rules (as in a term rewriting system [BN98, Ter03]).

- Typing and sorts:

  Erlang is dynamically typed and has very flexible typing rules; μCRL is strongly typed with a simple and restricted type system. Since we try to keep the specification in μCRL as close to the *Erlang* code as possible we construct in μCRL a data type *Term* in which all *Erlang* data types are embedded. The tool supports most *Erlang* data types: lists, integers, atoms, tuples, and records. However, the recently added *Erlang* bit-syntax implementing the data structure of bit sequences, is not considered by our tool.

  A consequence of creating the *Term* type is that `T` and `F` cannot be used as variables in *Erlang*, since they are used to define the μCRL type for booleans (they are reserved for that). Besides, integers in *Erlang* are based on the `Natural` type in μCRL, therefore, we are effectively restricting the integers to naturals.

  We base the sort *Term* on the sort *Boolean* and *Natural*, with their rather standard implementations. The following μCRL fragment shows the definition of some constructors and destructors:

```
sort
     Term
func
     int: Natural -> Term
     nil: -> Term
     cons: Term # Term -> Term
map
     hd: Term -> Term
     tl: Term -> Term
```

  The *Erlang* tuple {`reply,ok,[Client]`} is, for example, translated to the μCRL term

```
tuple(reply,tuple(ok,
                  tuplenil(cons(Client,nil))))
```

- Pattern matching in the data part:

  In Sect. 8.2.2.4, we have discussed the matching of function clauses and expressions for the functions that have side-effects. For the pure functions, the translation is much simpler. The header of function clauses can directly be

copied, since for the term matching on that level, matching in *Erlang* and μCRL are the same. We have to rewrite the body of the function clauses, where all statements have a fixed translation to μCRL rewrite rules. For example, an *Erlang* function with `case` statement

```
functionName (P1,P2,...,Pn) ->
           case E of
              Q1 -> E1;
              ...
              Qm -> Em
           end.
```

where `P1`, ..., `Pn`, `Q1`, ..., `Qm` are patterns and `E`, `E1`, ..., `Em` are expressions, is translated in several rewrite rules (where recursively the expressions are translated). The notation `var(P1,...,Pn)` stands for all variables in the patterns `P1`, ..., `Pn`.

```
functionName(P1,P2,...,Pn) ->
   case1(var(P1,...,Pn), E).

case1(var(P1,...,Pn), Q1) -> E1;
...
case1(var(P1,...,Pn), Qm) -> Em.
```

In this translation, `case1` stands for a function symbol uniquely chosen for the translation of every case statement. We can perform this transformation source-to-source and only in the last phase translate the *Erlang* code to μCRL.

An example of a real function could be the following one, which receives a concrete *Media Object* and a profile and checks if it is present in the list of pairs {`MO,Profile`} received as the third argument.

```
check_available_bw (MO, Profile, []) ->
    [];
check_available_bw (MO1, Profile,
                [{MO2,AvailableProfile}|OtherMOs]) ->
    case (MO1 == MO2) of
        true ->
            list_intersection (Profile, AvailableProfile);
        false ->
            check_available_bw (MO1, Profile, OtherMOs)
    end.
```

The translation to μCRL would be the following one:

```
check_available_bw(MO,Profile,nil) = nil

check_available_bw(MO1,Profile,
   cons(tuple(MO2,tuplenil(AvailableProfile)),OtherMOs)) =
case2(OtherMOs,AvailableProfile,MO2,Profile,
```

```
                                    MO1,equal(MO1,MO2))

case2(OtherMOs,AvailableProfile,MO2,Profile,MO1,true) =
   list_intersection(Profile,AvailableProfile)

case2(OtherMOs,AvailableProfile,MO2,Profile,MO1,false) =
   check_available_bw(MO1,Profile,OtherMOs)
```

Another statement with a similar translation is the *Erlang* match `P = E`. The way to deal with this statement is again to call a function and lift the match to the rewrite level. Functions with a match

```
functionName(P1,...,Pn) -> P = E, Expr.
```

are source-to-source translated to

```
functionName(P1,...,Pn) ->
   match1(var(P1,...,Pn), E).

match1(var(P1,...,Pn), P) -> Expr.
```

Note that, although the translation of these statements looks rather straightforward and easy, we slightly change the semantics in the translation. As long as we stay on the source-to-source level there is no danger, but a direct translation to μCRL would affect the behaviour of the program.

For example, *Erlang* uses priority rewriting, i.e. patterns are tried from top to bottom and if an expression matches a pattern, the other alternatives are not visited. In μCRL any matching rule could be taken. At the moment we therefore check that there are no overlapping patterns in the definition, but in fact, one should rewrite the patterns to a non-overlapping set.

Second, the *Erlang* process that evaluates a match crashes if none of the patterns matches the given expression. In μCRL the rewriting of a term stops when no matching rewrite rule is found. The term is considered to be in normal form, even if it is not a closed term with only constructors in it. Normally, such terms cause problems later in the generation of the state space, where the state space generator crashes, but that is not a nice way of finding bugs in *Erlang* code. Moreover, a crash in an *Erlang* process might be the intention of the programmer and need not necessarily be a bug.

In our translation tool we have not solved those two incompatibilities in semantics. Instead, we concentrated on *Erlang* programs where these problems would not occur.

An example of the match for a real function would be a slightly modified version of the previous example, introducing a match sentence:

```
check_available_bw (MO, Profile, []) ->
    [];
check_available_bw (MO1, Profile,
```

```
          [{MO2,AvailableProfile}|OtherMOs]) ->
    ListIntersection =
       list_intersection (Profile, AvailableProfile),
    case (MO1 == MO2) of
        true ->
            ListIntersection;
        false ->
            check_available_bw (MO1, Profile, OtherMOs)
    end.
```

And the translation to μCRL:

```
check_available_bw(MO,Profile,nil) = nil
check_available_bw(MO1,Profile,
      cons(tuple(MO2,tuplenil(AvailableProfile)),OtherMOs)) =
    match2(OtherMOs,AvailableProfile,MO2,Profile,MO1,
                  list_intersection(Profile,AvailableProfile))

match2(OtherMOs,AvailableProfile,MO2,
                        Profile,MO1,ListIntersection) =
    case3(ListIntersection,MO1,Profile,MO2,
                     AvailableProfile,OtherMOs,equal(MO1,MO2))

case3(ListIntersection,MO1,Profile,MO2,
                  AvailableProfile,OtherMOs,true) =
    ListIntersection
case3(ListIntersection,MO1,Profile,MO2,
                  AvailableProfile,OtherMOs,false) =
    check_available_bw(MO1,Profile,OtherMOs)
```

8.2.2.9 Module system

Erlang code is divided into modules, each module consisting of a sequence of attributes and function declarations. Process algebras on the contrary, do not have module systems, although some tools (e.g., the CADP tool set [GLM02] for LOTOS [ISO88]) support a module system.

To prepare the conversion of the given collection of *Erlang* modules into one μCRL specification, we perform a source-to-source transformation. Every call to a function *f* is replaced by the *Erlang* qualified call *modulename:f*, where *modulename* is the name of the module where the function *f* is implemented.

Some modules in the standard library are translated once and for all to μCRL and the code of those functions is simply linked in at translation time. For the other functions, we assume all necessary modules given and change the name of the function definition and function call to the same name, viz. *modulename_f*, in the μCRL translation.

Somehow it is strange that a relative modern specification language has so poor features for specifying large software systems on a high level. The language LOTOS is better in this respect, but also in that language support for higher-order functions, for example, is lacking.

8.2.3 Overview of the etomcrl tool

In Sect. 8.2.2.1 – 8.2.2.9, we have described the main issues of the translation from *Erlang* to μCRL. Now, in this section, we describe the architecture of the translation tool, called `etomcrl`, in which we clarify the order in which the steps described before are taken.

The tool is completely implemented in *Erlang*, and takes as input the source code of the supervision tree and generic server based application and generates a file with the correspondent μCRL specification.

The module called `etomcrl` is the main module of the tool implementation. The `supervisor` function inside that module starts the compilation process, that takes as input arguments: the module, function and arguments of the supervision tree behaviour in charge of starting the application we want to translate. Therefore, in order to start the translation, the way of invoking the tool is exactly the same as for starting the supervision tree, but obviously calling to a different module.

Fig. 8.4 shows the three main phases in the `etomcrl` tool. First, a source-to-source transformation is performed on the level of *Erlang*, resulting in *Erlang* code that exhibits the same behaviour to an observer as the original code, but is optimized for the translation. Second, the side-effect-free part of the code is separated from the part with side-effects, since the translation is different for each of the two parts. Third, the translated files are combined into a single μCRL specification.

- `etoe`: the first phase of the transformation can be seen as a preprocessor, that performs some *Erlang* to *Erlang* source code transformations. The main changes are: the supervision tree is partially evaluated in order to extract the processes of the system, as introduced in Sect. 8.2.2.6. The `lower` module is used to remove higher order functions as explained in Sect. 8.2.2.7. The `noio` module removes the calls to the input/output module; for the purposes of verifying embedded systems the messages to the standard output are not relevant, and can be discarded. Finally, the code is analyzed and split in two different parts: the side-effect-free part, with only pure computations, and the side-effect part. These parts are going to be processed in different ways in the next stage, as explained in Sect. 8.2.2.3. Candidates to μCRL actions are already detected and provided as output of this phase.
- `etopa`: the second phase of the translation is from *Erlang* source code to an internal representation very close to the process algebra syntax and semantics. The main transformations carried out are: `gs_replace` changes *Erlang* gen_server related code as it is explained in Sect. 8.2.2.2 in all the call-back modules implementing the generic server behaviour. Some of the built-in-functions are also redefined in order to use the predefined ones already included as part of the `etomcrl` tool instead of translating them from those included in the application to be translated. The `locals` module takes care of encoding the modules into the function names, as explained in Sect. 8.2.2.9. After these changes, the code is

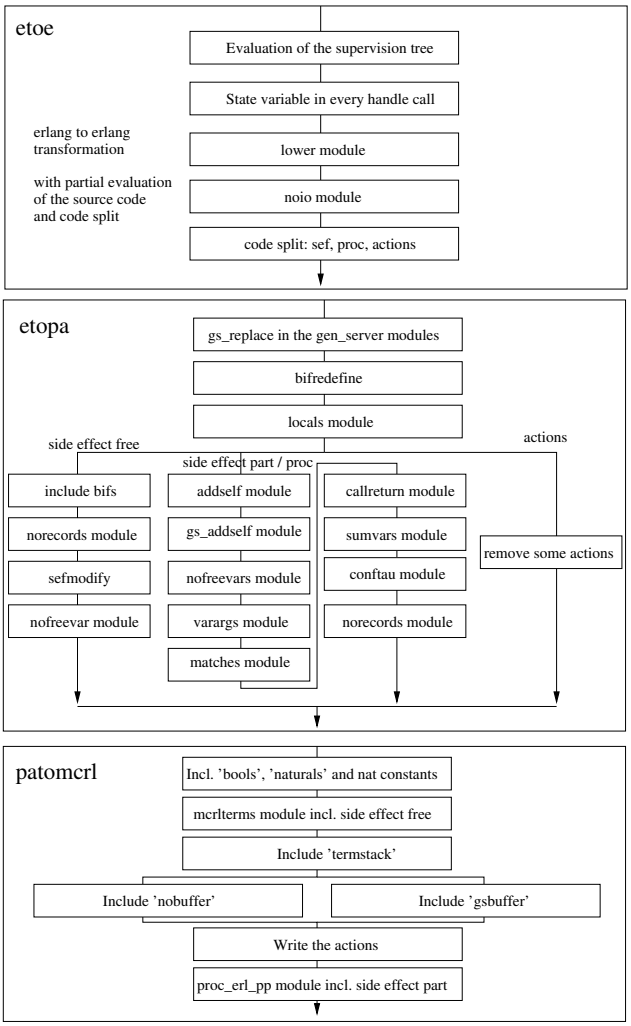


Figure 8.4: Architecture of the `etomcrl` tool

split into the three parts already identified in the `etoe` phase of the translation.

* For the side-effect-free part of the code, the following translations are performed: some *Erlang* library functions are included for translation; records are translated to a data structure that can be defined inductively. The module `sefmodify` changes the function clauses related to the matching problem explained in Sect. 8.2.2.8. The module `nofreevar` replaces the underscores (which are used for meaning 'anything' in a pattern matching and are not supported in μ`CRL`) in the *Erlang* source code to uniquely chosen free variables.
* For the side-effect part of the code, the following translations are performed: the `addself` and `gs_addself` modules change the code such that the process identifiers can be used as arguments, as explained in Sect. 8.2.2.1. The `nofreevars` module is also applied to this part of the code for the same reason as above. The modules

varargs and matches are performing the transformations explained in Sect. 8.2.2.4. The callreturn module introduces the stack explained in Sect. 8.2.2.3. The module sumvars is introducing the *sum* construct, as explained in Sect. 8.2.2.3. Finally, conftau introduces some extra steps needed by the μCRL toolset (adds extra *tau* step to recursive calls of function without progress), and the records are also removed from this part of the code.

  - For the actions part of the code, only some of then, related to the introduction of *bifs* and the user defined actions, are changed or removed.

- patomcrl: the third phase is a back-end for translating the internal, process algebra, representation to μCRL. The main steps in this phase are: the code for the standard inductive definitions for the μCRL sorts bools and naturals are introduced. The module mcrlterms translates the side-effect-free part to μCRL syntax. The buffer (if present, because the tool also allows the user to select a translation without buffer where all the communication is synchronous) and stack have standard μCRL implementations that are inserted. The user can decide to create a special translation without the buffer where all the communication is synchronous between any pair of processes; this can be useful in the verification process for experimenting with the generation of smaller state spaces, where some properties should still hold, and therefore some errors can already be detected in the software. Actions are inserted as communication actions in the μCRL specification, and finally, proc_erl_pp translates the side-effect part from the internal notation to μCRL syntax.

The tool could be reused for other kind of transformation, e.g. if we want to extract a LOTOS [ISO88] specification, we only need to write a new back-end for translating from the internal representation to the LOTOS syntax. Therefore, even though the tool has been built for a quite concrete purpose, its main ideas can be reused for similar approaches.

8.2.4 Detecting messages matching a given pattern

In the translation process, sometimes it is interesting to be able to introduce a semantic modification in the specification so that some of the paths are removed in the state space that is going to be later generated from μCRL. In some situations, we are only interested in generating the state space until a given event takes place, and then stop the graph generation from that state, creating some kind of deadlock. Any property that can be verified against this simplified graph could also be model checked against the complete one, but this way we can simplify the properties to be defined. Also, the generation of a smaller state space is faster, which can be a key advantage if we are dealing with quite complex software.

In *Erlang* source code, it would be very useful to be able to specify somewhere in each generic server module which messages we want to *observe* in the generated graph. We would change therefore the translation mechanism for the introduction of extra μCRL code according to that messages that need to be observed. A powerful

way of doing this is using *Erlang* pattern matching, allowing the user of the `etomcrl` tool to provide a list of patterns to messages. Moreover, sometimes several groups of messages want to be *observed*, each group having a different meaning and being used in a different way later for verification purposes.

The solution we propose is letting the user of our tool to specify *observe messages* as lists of patterns associated to a *message type*. They can be declared using a special *Erlang* header (the compiler ignores those kind of user defined headers) as follows:

```
-observe_messages({message_type1,["{_,_,tag1}"]}).
-observe_messages({message_type2,["{_,tag2,_}","{tag3,_,_,_}"]}).
```

Any message matching {`_,_,tag1`} will be tagged in the state graph as a message of type `message_type1` and will originate a deadlock state. Now the question is, how do we include a modification in the specification so that the graph we want is generated.

A new module for the `etomcrl` compiler was developed in order to perform a source to source transformation. The transformation adds a message checking in those generic server modules that have the above explained observe message headers. It is called in the `etoe` part of the compiler described in Fig. 8.4, before any other source to source transformation is carried out.

The module receives the implementation of a generic server and, after removing the headers where the message patterns to be observed are specified, a new function `observe_messages_check/1` is added to the new source code. This function receives a message and returns true if it matches any of the patterns specified in the `observe_messages` attribute.

All the reply messages (the tuples {`reply,Message,NewState`}) that appear in the `handle_call` functions are then substituted by:

```
case observe_messages_check(Message) of
    {true,OM_MessageType} ->
        observe_messages_show(OM_MessageType,Message),
        {reply,Message,NewState};
    {false,_} ->
        {reply,Message,NewState}
end;
```

Where `observe_messages_show` is a function that will be translated as a built in μ`CRL` function. And the implementation of `observe_messages_check` would be automatically generated from the *observe messages* header as follows:

```
observe_messages_check(OM_Message) ->
    case OM_Message of
        {_,_,tag1} ->
            {true,message_type1};
        {_,tag2,_} ->
            {true,message_type2};
        {tag3,_,_,_} ->
            {true,message_type2};
        _ ->
            {false,notype}
    end.
```

After this changes, all the code is translated to μCRL as explained in the previous sections, and with the only exception of `observe_messages_show`, that produces the following μCRL built in code:

```
act
   omshow: Term # Term

proc

observe_messages_show(Arg1:Term, Arg2:Term) =
          omshow(Arg1, Arg2). delta
```

The action `omshow` will allow us to visualize in the graph the message type and the actual message, and the `delta` is creating the deadlock node and stopping the graph generation

If no `observe_messages` patterns are specified, no changes are made to the original module.

During the creation of the state space, the generation is automatically stopped, due to the delta action, as soon as one of these special messages are reached. This is an easy way of seen when is the first time that those messages happen, and can be very useful when, for example, looking for system bottlenecks. In Sect. 9.2.3.3, the use of this technique in order to find bottlenecks in the *VoDKA* system is illustrated.

8.2.5 `arch_graph`: inter-process relations from the state graph

Apart from the core compiler from *Erlang* to μCRL, `etomcrl` also includes some complementary tools, like one for visualizing the supervision tree of the code to be translated. This tools help in order to use `etomcrl` in a real environment and understand easier what is going on and why the code is translated in a given way.

While using `etomcrl` in the context of the *VoDKA* server, the need for an extra tool of this kind was detected. The state graphs generated from the μCRL specification are formed by nodes representing global states and arrows representing messages. The information about all the messages sent between any pair of processes is, therefore, inside the state graph.

A module for the `etomcrl` tool was developed in order to be able to extract software architecture information from the state graph of the system.

It takes a state graph file as input and generates a new graph where the nodes are the processes in the system architecture and the messages, the same ones as in the original graph. The labels in the input graph are assumed to contain the process source and destination for every message. Identity transitions and assertions are removed. The tool returns a new AUT file with numbered states that represent the processes and transitions that represent the messages. It also returns a dictionary associating the process identifier or process name in the *Erlang* source code with the number in the AUT file, in order to increase the possibilities of tracing information back to the original model. Repeated messages between processes are removed, so that only one unique transition is created for each message expression exchanged between two processes. If the users of the tool are interested, they can define an

abstraction function (similar to a hash function) for grouping messages according the results of that function.

Examples showing how this module is used in the context of the *VoDKA* project are given in Sect. 9.2.3.2.

8.2.6 Conclusions and limitations

In this section, we described how a functional language with support for concurrency and distribution can be translated to a process algebra. The ingenuity of the translation shows in the choices we made for mapping concepts of one language to concepts of the other. For example, we make strong use of the design patterns in *Erlang* to enable a smooth translation. By translating *Erlang* to μCRL we can use formal verification tools developed for μCRL and labelled transition systems.

Other approaches to the formal verification of software that share some ideas with the one we have presented include the specification language Promela and model checker SPIN [Hol91], PathFinder [HP00], and Bandera [HD01]. In the first case Promela is very close to C while the targeted language for the latter two is Java. Relevant tools developed for *Erlang* include a theorem prover with the *Erlang* semantics built into it [FGN$^+$03, Fre01] and the model-checker of Huch [Huc99], which works on code directly.

The theorem prover can in an inefficient way be used to symbolically explore part of the state space. Its power is though in interactive proofs of a different nature, whereas the model checking approach is efficient and automatic. Huch's approach differs from ours in the way he abstracts data aspects which we consider crucial. In particular, he abstracts *case* statements by non-deterministic choices, losing all reference to the data.

The tool that we constructed to perform the transformation has been evaluated by two major case-studies of which the results are reported elsewhere [ABD04, AS02]. The tool allows us and others to apply formal verification tools on real industrial code. It has been written in such a way that it is not μCRL specific, but can easily be ported to other process algebras or similar approaches.

Before and during the development of the tool, we have repeatedly asked ourselves whether it would be better to build a verification tool directly on the level of *Erlang* instead of translating *Erlang* to a process algebra. At that point we decided to build a translation tool and use all the research done over the years by other groups, instead of concentrate on doing the research ourselves and get only part of all theory implemented. In this way we benefit from years of experience with building verification tools and optimizing those tools and pay the minimal price of having to write a kind of compiler ourselves. However, since any formalism that we want to map our language to has restrictions, the question re-appeared several times. This alternative approach of generating the state space directly from *Erlang* is studied in Sect. 8.5.

We identified three main restrictions in verification formalisms that we considered. First, specification languages lack the support in the development tools that modern programming languages have. A simple thing like a debugger or a way to write code in modules instead of one big specification are often missing. Second, programming languages have powerful constructs both in statements and in data structures, e.g. higher-order functions, list comprehensions, records, inher-

itance. These constructs are seldomly supported by specification languages, which most of the time remind of languages from the early eighties. Third, specification languages have poor and inefficient support for arithmetics. They are designed for specifying problems. Rather soon people realized that for a lot of interesting properties, one has to take data into account in the specification. Therefore, specification languages also need to be able to perform some computations. However, the implementation of the computations plays a minor role. Hence, a tool to create a rather small state space can still spend an amazing amount of time in just performing simple arithmetic.

The `etomcrl` tool assumes the *Erlang* code received as input is correct both from a syntactic and semantic point of view. This means that for code with syntactic or semantic errors the correctness of the μCRL output is not guaranteed. By assuming some kind of semantic correctness we mean here that if the code includes things like a call to a function whose value is not used, an incorrect translation can be generated. We took this design decision because there are other tools available for checking the correctness of the *Erlang* code, and we wanted to concentrate in creating a simpler and faster compiler.

As a small comment, we can underwrite the conclusion of Lamport and Paulson [LP99]: specification languages should not be typed. At least, if one translates a programming language to a specification language, a type system is often in the way. The programming language has certainly a type system and hence the types need not be checked on a specification language level. Moreover, the type system of a modern language is easily incompatible with the type system of the specification languages around. Hence, the types get in the way when translating.

Despite some limitations in the process algebra languages, the tools developed for them (e.g. [Wou01, GLM02]) make a translation very rewarding. The time it takes to create a state space of a reasonably complicated system or the time necessary for model checking some properties has never been a restriction in our case-studies. We have been able to verify several properties of real code with a reasonable complexity. Without counting the about 2000 lines of code that are given as design patterns and library code, our case studies consisted of a few hundred lines of code. From the experiment we can conclude that this verification approach scales to larger size examples. We hope to be able to improve the integration of several tools in order to make source code verification even simpler in the future.

We explore more in detail the possibilities of `etomcrl` in the next chapter, when describing in detail the experiment carried out for this thesis.

8.3 μCRL `toolset`: generating the state space from the process algebra

In Sect. 3.3 we introduced the main concepts related to process algebras. We also explained that μCRL [Gro97] is a process algebra where the specifications have two parts, one representing the data part, including type constructors and rewriting rules, and the other representing the processes, communication with synchronization actions. With the `etomcrl` tool, we are able to generate automatically μCRL specifications from *Erlang* source code. One of the reasons why we selected μCRL as the target specification language, apart from the features of the language itself,

was the availability of the μCRL `toolset` [SEN99, Wou01].

8.3.1 Introduction and motivation of the tool

The μCRL `toolset` is a set of tools developed in the Center for Mathematics and Computer Science (CWI) of Amsterdam, for analyzing specifications of concurrent systems, described using the μCRL language. The tools support a restricted part of the μCRL language, the linear process operator format (LPO).

The main motivation for selecting the tool is obviously very associated to the fact of selecting the μCRL specification language. In concrete, the μCRL `toolset` is the result of the research carried out by a team at the CWI during about a decade, and we had the intention of being able to reuse that potential in our approach, taking advantage of the different tools provided.

The main components included in the toolset are:

- `mcrl`: performs two steps, first checking if a given μCRL specification is well-formed, and then trying to linearize it. In order to check if the specification is well-formed, the tool checks things like the syntax, the typing of the terms, conditions and equations, the existence of only one initial process, or checking that communication actions behave in associative and commutative way. The output can be either binary or in a text-based readable format.

- `msim`: interactive simulation of a μCRL specification. Allows the user to execute step by step a given simulation.

- `instantiator`: generates a state space from a linearized μCRL specification. Takes as input the output of `mcrl` in binary format and can generate different output formats for the state space, like SVC or AUT.

- `pp`: pretty printer of a linearized μCRL specification.

- Specification optimization tools:

  - `rewr`: normalizes a linearized μCRL specification.
  - `constelm`: removes from a linearized μCRL specification the data parameters that are constant throughout any run of the process.
  - `parelm`: removes froma linearized μCRL specification the data parameters and sum operators that do not influence the system behaviour.
  - `structelm`: expands the composite data types of a linearized specification.
  - `sumelm`: substitutes sum expressions by a concrete term when it is possible.

Fig. 8.5, from the μCRL `toolset` documentation, describes in a very illustrative way the collaboration between the different pieces included in the μCRL `toolset`.

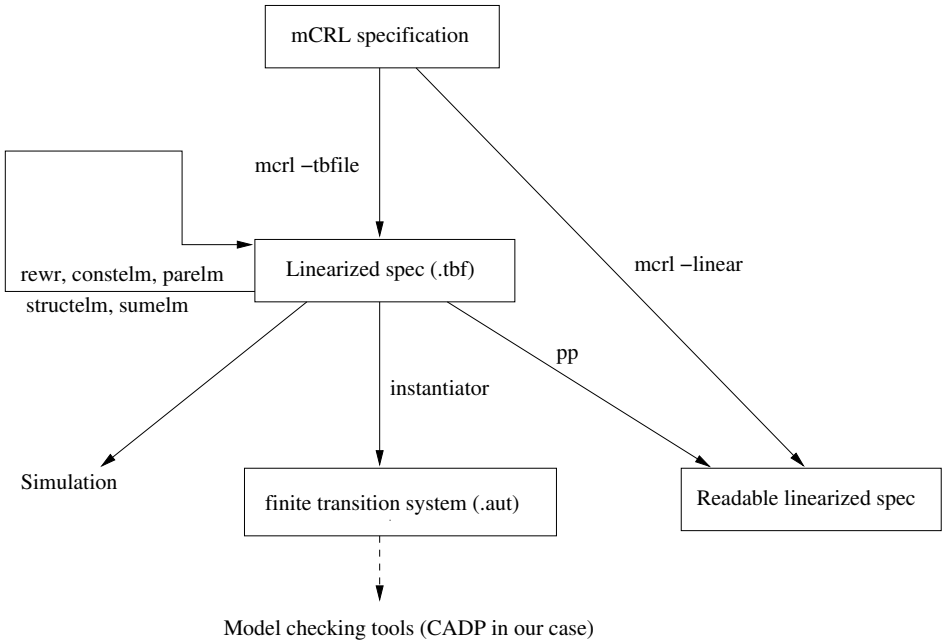


Figure 8.5: Collaboration of the tools included in the μCRL toolset

8.3.2 Using the μCRL toolset for our purposes

In the next chapter we will explain the experiments in which we have used the μCRL toolset. We have made extensive use of `mcrl` and `instantiator` for generating all kind of state spaces from different *Erlang* programs. We have carried out some experiments with the optimization tools in order to simplify the specification for generating a smaller state space, obtaining good results.

We generated AUT files in order to provide them to `CADP`. The AUT files are very simple. They are text files where the first line describes the state space (including the initial state, number of states and number of transitions) and the rest of the lines are just triples representing transitions (source state, name of the transition and destination state). These are very well handled by `CADP`. The disadvantage is that they are less efficient than other formats, which can cause a slower state space generation for really big state spaces.

In general, the results were very satisfactory, but we have found mainly three issues or limitations in our experience with the tool:

- The error messages of the `instantiator` are quite cryptic. During an important part of the research, we have been using the μCRL toolset at the same time we developed and improved `etomcrl`. When there were errors in the translation, the generated μCRL specification was sometimes syntactically or semantically incorrect. When the error was syntactic, it was detected by `mcrl` before linearizing the specification and was quite easy to fix. In the cases where the error was semantic, the state space generation failed, and the error messages produced for the user were often referring to some internal details of the generation processes, making very difficult to trace the error back to the `etomcrl` translation and to the original *Erlang* program. This was,

therefore, an obstacle in our goal of making all the steps of the verification process traceable back to the original source code.

- The efficiency of the `instantiator` is an issue. For complex configurations of our input *Erlang* program, the translated specification was also quite complex, producing very big state spaces. The fact of using AUT format for the output files reduces the efficiency of the generation; this could be avoided by using other formats that have been added to the last versions of the toolset, and it is left as future work. Besides, the representation in μCRL of the natural numbers, based on the recursive definition with the constructors *zero* and *successor*, makes any kind of computation with big numbers extremely inefficient; in order to avoid this, be have tried to abstract from big numbers in our models, but a more optimal and flexible solution would imply the implementation inside `instantiator` of a mechanism for moving complex computations outside of μCRL.

- Optimization tools are complex to use and not completely automatic. The tools for optimizing the specification described above are quite powerful and allow the user to simplify the specification and therefore the final state space. But due to their theorem proving nature, they are not very automatic and require a human to participate interactively in the process. This goes a bit against our global approach of going automatically from the system architecture to the extraction of properties, and it is one of the reasons why we have not used them more extensively. Also, the lack of a guidance or real examples on how to use them for reducing real state spaces makes their usage quite challenging.

8.4 CADP: model checking the state space

8.4.1 Introduction and motivation of the tool

In previous sections we have seen that our generic approach consists in translating from the original source code in *Erlang* to the process algebra specification in μCRL, in order to be able to generate the state space. But the state space itself is not very useful (for real examples it is not even visible or printable due to its enormous size) if we do not have the right tools for manipulating and analyzing it.

For that purpose we decided to use a set of tools, the CÆSAR/ALDEBARAN Development Package (CADP) [FGKM96, GLM02].

CADP is a complete toolbox for specifying and verifying asynchronous finite-state systems described using the process algebra LOTOS [BB87]. The toolbox includes different kinds of tools oriented to help the user in different stages of the system specification, prototyping, development or verification. CADP has been used for years both by industry and academia, and it has been applied in a considerable number of case studies.

From the above description, it is clear that CADP is not only a finite state analyzer and checker. It has a wider functionality, but in our case we have mainly used the tools related to label transition system manipulation and model checking.

8.4.2 Parts of the CADP that we are using

The parts of the CADP toolbox we use in our method are:

- ALDBARAN. A tool for verifying distributed systems represented by a state space. It allows the reduction of the state space modulo equivalence relations such as strong bisimulation, observational equivalence, branching bisimulation, and so on.
- BCG (Binary Coded Graphs). We use AUT files generated by the μCRL toolset as input for CADP, but then we translate them inside CADP into BCG files. They have a binary format which is more optimal and can be up to 20 times smaller than AUT. Together with the format itself, CADP provides a set of tools for opening and manipulating BCG files.
- SVL [GL01a] (Script Verification Language) is an scripting language for making the verification easier. It offers operators for generation, minimization, label hiding, label renaming, abstraction, comparison and model checking of state spaces. CADP provides a compiler from SVL to a UNIX script that performs all the steps (calls to different CADP tools) needed for perform the verification.
- BGC_MIN. Tool that complements and improves ALDÉBARAN and provides minimization algorithms for state space graphs encoded using the BGC format. It only implements strong and branching bisimulation, but it can handle bigger state spaces and it is faster reducing.
- EVALUATOR. Provides on-the-fly model checking of regular alternation-free μ-Calculus formulas for the input state spaces. Regular alternation-free μ-Calculus is an extension of the alternation-free fragment of the modal μ-Calculus with action predicates and regular expressions over action sequences.
- EUCALYPTUS, shown in Fig. 8.6, is a graphical user interface developed in Tcl/Tk that provides an homogeneous entry point for all the tools included in CADP.

In our method, that will be explained in the next chapter, we use CADP in order to reduce and simplify a state space, and later model checking it.

A typical SVL script for reducing a state space using a combination of CADP tools would look like:

```
"abstract.bcg" = total hide all but ".*pid(0).*" in
                     "vodka.aut";

"state_space.bcg" = safety reduction of
                     total rename using "simplify.rename" in
                     total hide "assertion.*", "buffercall.*" in
                     total hide "reply(pid(0),[^{].*)" in
                     total hide "call(.*lookup,.*)" in
                     total hide ".*lookupAns,.*,\[{.*\].*" in
                     "abstract.bcg"
```

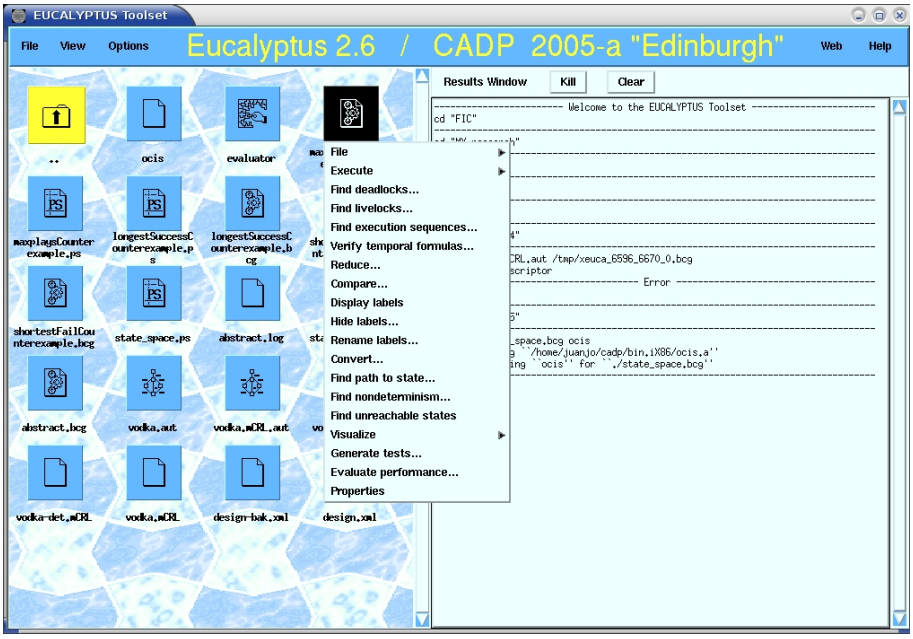


Figure 8.6: The EUCALYPTUS GUI integrates all the CADP tools

The script would take `vodka.out` (an AUT file with the state space for a given configuration of the *VoDKA* software) as input. It first would hide (convert in internal actions) all the labels but those one matching the given regular expression and then would convert the state space to BCG format. After that, another set of hiding and renaming actions are carried out and then a new reduced graph will create another BCG file.

The `simplify.rename` file would contain renaming expressions for simplifying the labels. In this example, the messages used in the *VoDKA* protocol are renamed tu much simpler `play` and `fail` tags:

```
rename
   "call(.*,{play,\(.*\),\(.*\)},.*)" -> "play(\2,\1)"
   "reply(.*lookupAns,\(.*\),\[\].*)" -> "fail(\1)"
```

In order to model check μ-`Calculus` properties against a BCG file containing a state space, we need to first write the formula in a text file. A very simple formula for a state space where the transitions are formed by an action name `play` with three arguments (process name, movie and bandwidth) would be:

```
['play(.*,m1,2)'.'play(.*,m1,2)']
        <'play(.*,m2,2)'.'play(.*,m2,2)'>true
```

The formula says that, starting at the initial state, all transition sequences that have two play actions for movie `m1` with bandwidth 2, are leading to a state where at least exists one transition sequence where there are two play actions for movie `m2` with bandwidth 2. In other words: after playing twice movie `m1` with bandwidth 2, it is still possible to play `m2` twice with the same bandwidth.

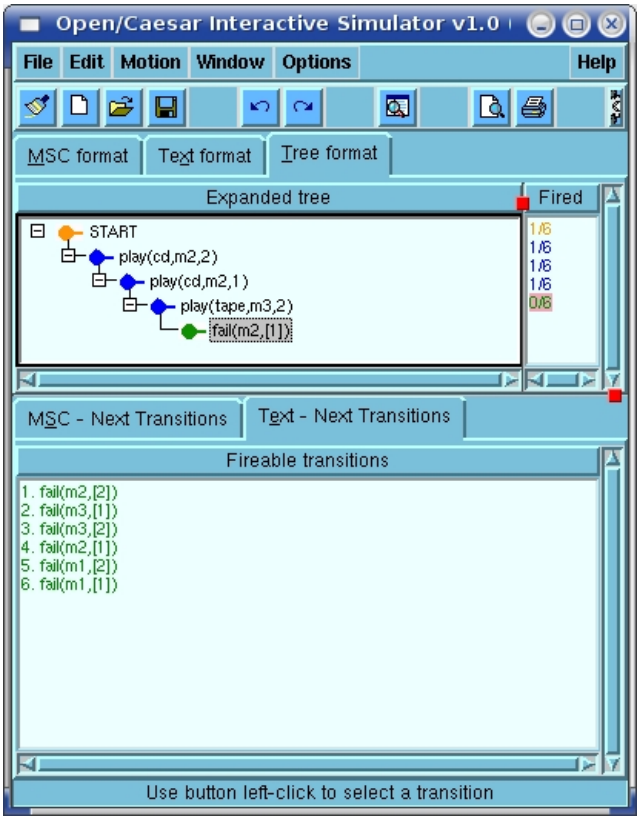


Figure 8.7: CADP includes an interactive state space simulator

In order to check this kind of formulas, we can use text-based commands similar to the following one:

```
bcg_open state_space.bcg evaluator -diag conterexample.bcg \
                                   -verbose property.mcl
```

The μ-Calculus formula is given inside the file `property.mcl`, and if the condition is not true, a counterexample is returned containing the part of the graph that shows an example where the property does not hold.

All the previous tools are integrated by EUCALYPTUS under the same graphical user interface. Besides, other minor tools for exploring the state spaces (getting the number of states, the list of labels), or for simulating the execution of the state space (shown in Fig. 8.7) interactively are provided to the user. We have made extensively use of all the interfaces and tools while developing our method and applying it to our case study.

8.5 McErlang: model checking the state space directly from *Erlang*

The approach of going from *Erlang* to μCRL and then reusing the research carried out during a decade (μCRL toolset and CADP) has some obvious advantages, as

stated in the previous sections. But, although they have similarities, *Erlang* and μCRL still belong to two different worlds (high level programming languages and process algebra specifications), and the translation itself, together with the fact of using several different tools, are including some limitations to the approach.

From the beginning, we thought that exploring the other alternative, a possibly less mature but more *ad hoc* tool for generating the state space and doing model checking from the *Erlang* model, was interesting.

McErlang [FBE06] is a new tool currently under development by Lars-Åke Fredlund at the Universidad Politécnica de Madrid (UPM), which takes as input an *Erlang* program and checks it against safety correctness properties also specified as Erlang programs. As the verification of a program uses on-the-fly model checking techniques, it is sometimes not required to explore the whole state space of the Erlang program. In this section we explain the fundamentals of the tool.

8.5.1 Introduction to the tool

The main goal of the McErlang tool is to implement a model checker for *Erlang* programs. The input to the model checker is potentially any kind of *Erlang* program together with an special *Erlang* call-back module specifying the behavioural property to be checked (called the *monitor*), an implementation of the *state table* that determines when a new state has already been explored, and a set of state *abstractions* also expressed in a call-back module. The output can be either a positive answer saying that the property holds for all the generated states, or a negative one together with a counterexample, showing that the program does not possess that property. The tool can also be used to generate the state space and visualise it; this could permit us the usage of other model checking tools like CADP for the actual property checking.

The idea behind the McErlang implementation is to replace part of the *Erlang* runtime system which implements concurrency and message passing, while still using the runtime system for the evaluation of the sequential part of the input programs. The relevant events in the software, mainly the ones having side effects (process creation, message sending, process linking, etc.), are substituted in the original source code by a call to special modules that are part of the model checker. Receive statements (and other special side-effects) are changed so that instead of actually waiting for the message, a special value is returned which indicates that the process is waiting. This value is a structure which contains information about what to do when the message is actually received in the model checker's simulation of the process (it contains one function for checking if the message is receivable, and another to continue the evaluation once the reception occurs).

The model checker has a complex internal state in which the current situation of the runtime system is represented. The structure that is maintained by the model checker includes the list of processes that have been created and are still alive, together with the state of each of them (mailbox, pid, process state, and so on). Moreover the global state kept by the model checker runtime system includes a structure to record process links, information about registered process identifiers, etc.

The model checking of an Erlang program using McErlang requires the following steps:

- First, the original source code of the system is translated into the internal language, where all the side-effect operations are converted into calls to the model checker modules or special return values. There is an ongoing result to automate this translation, but it was not ready in time for the experiments reported in this thesis.

- Second, the model checker starts the *Erlang* program by calling a designated start function (using the `apply` *Erlang* function). The function call either returns a normal return value, signalling that the associated process has finished and can thus be removed from the global model checker state. Alternatively, the function call returns a special return value signalling that a receive statement is pending. At that time the model checker can schedule another (simulated) process for execution.

  Other side effects, such as e.g. spawning a new process, are handled by modifying the global state of the model checker (the process table) during the call to `apply`. Once the function call terminates, the newly created process can be run by the model checker.

  That is, instead of creating new processes or actually sending messages between them, the model checker global state is updated simulating those actions.

- Third, a depth-first algorithm is used in order to explore all possible future states of the system. Multiple next states are possible during the state-space exploration for obvious reasons. First, multiple processes may be scheduled for execution, and second, the tool user may have deliberately specified a non-deterministic choice in the verified program.

- Whenever a process does a computation step (i.e., typically starting by reading a value from the queue and finishes waiting in another receive statement), the target state is analysed with regards to two checks:

  - The *monitor* is invoked to check whether the correctness property holds of the combination of the target program state and the old monitor state. If successful, the monitor returns an updated monitor state.

    Correctness properties can be implemented, therefore, as finite state machines where depending on the current state some actions or state values are accepted and others are not.

  - A call is made to the *abstractions* module for abstracting state spaces or transitions, and with the result a call is made to the *hash table* in order to determine if a given state has been seen previously or not. The user of the tool can define *ad hoc* abstractions where the irrelevant information is not taken into account at the time of distinguishing states. Adapted hash tables can also be defined, including the possibility of stopping the generation of new states in a given path depending on the state of the model checker. Two states are the same if the abstractions point to the same element of the hash table, taking into account both the state of the monitor and the state of the model checker.

8.5.2 Internal implementation of `McErlang`

8.5.2.1 The internal language

All the side-effects in the original source code need to be translated into a call to a function implemented inside the model checker modules. For example, sending a message is done with `evOS:send()`, which updates the global state of the monitor adding the message to the queue of the receiving process.

The possible values that a function can return are:

- `{recv, {Mod, Fun, Args}}`

  It substitutes a receive statement in the original source code. The second element of the tuple represents the checking message function that should be called once a message is received to determine if it can be read. The function `Fun` is going to be invoked using `apply(Mod,Fun,[Value,Args])`, where `Value` is a message in the process queue, and `Args` are the list of values from the context that are going to be needed later on.

  The function can return two possible values. It returns false if the message in the queue is not readable (it does not match), and true `{true, ContFun}` when the message can be read. `ContFun` is the anonymous continuation function that is going to receive two arguments: the message received and the information contained in `Args`.

  When the model checker evaluates a process that returns this kind of `recv` value, it places it in a blocked state. It will be selected again by the model checker for evaluating the checking message function once a message is received. If the return value of that function is positive, the state of the process will be updated to state *receivable*. Next time it is selected by the model checker, the message is actually removed from the queue and the continuation function is evaluated. This way, we can observe while model checking three different process states when receiving a message, which can be interesting for some properties to be checked.

  The following *Erlang* source code:

```
-module(ex).

...
Value1 = 10,
Value2 = 20,
receive
   {message, Msg} ->
           function1(Value1, Msg),
   ok;
   nomessage ->
           function2(Value2),
           error
end.
```

  Would be translated including the `recv` return value, a function for checking the message and a continuation anonymous function, as follows:

```
-module(ex).
...
-export([checking_function/2]).

...
Value1 = 10,
Value2 = 20,
{recv, {ex, checking_function, {Value1, Value2}}}.

checking_function({message, Msg}, {Value1, Value2}) ->
    {true, fun ({message, Msg}, {Value1, Value2}) ->
                function1(Value1, Msg),
                ok
           end};
checking_function(nomessage, {Value1, Value2}]) ->
    {true, fun (nomessage, {Value1, Value2}) ->
                function2(Value2),
                error
           end};
checking_function(_,_) ->
    false.
```

- {`choice, ListOfFuns`}

  Although in *Erlang* a non-deterministic choice construct is not available, when doing model checking of *Erlang* programs it is often useful to have that kind of expressiveness.

  Whenever a function returns a `choice` value, the model checker will explore all the possible paths corresponding to the evaluation of each of the functions in the list. Each function should be defined as a triple of the form {`Mod,Fun,Args`}, where the arguments are the values from the context that are going to be needed in the continuation.

- {`pause, {Mod, Fun, Args}`}

  It is similar to the previous return value, but now only one choice is given for the continuation. The pause is interesting for interrupting the execution of a program in order to explicitly make visible some states for the monitors.

- {`letexp, {Expr, {Mod, Fun, Args}}`}

  The `letexp` construct is needed when a receive statement occurs inside another Erlang expression. When `Expr` evaluates to a non-special value `Value`, the function {`Mod,Fun`} will be called with two arguments: `Value` and `Args`.

  In the previous example, if the receive statement occurs inside a send, we need to surround the `receive` with a `letexp` expression:

```
Pid ! receive
          {message, Msg} ->
    function1(Value1, Msg), ok;
          nomessage ->
    function2(Value2), error
      end
```

We can use `letexp` in the following translation (only the new parts are shown):

```
-export([cont_fun/2]).

...
{letexp, {{recv, {ex, checking_function, {Value1, Value2}}},
          {ex, cont_fun, {}}}}.
...

cont_fun(Value, {}) -> Pid!Value.
```

McErlang has built-in support for using some *Erlang* behaviours. This makes easier to perform the code translations, as the code of the behaviours is already adapted and translated, being part of the tool.

This means that calls to modules like `supervisor` or `gen_server` need to be translated into calls to `ev_supervisor` or `ev_gen_server`.

As the generic server `call` function returns as a result the message sent back from the server, it needs to be handled similarly to the `receive` statement. The following code using a generic server call:

```
...
{lookupAns, NewOptions} = gen_server:call(Driver,MessageSent),
function1(NewOptons, MessageSent),
...
```

Would be translated as follows:

```
ev_gen_server:call(Driver,MessageSent,
                   {?MODULE,continuation_function,{MessageSent}),
...

continuation_function({MessageSent},{lookupAns, NewOptions}) ->
    function1(NewOptions, MessageSent).
```

Internally, the continuation function is used to construct the `recv` message that is part of the implementation of `call` (it first sends a message and then waits until it is received).

McErlang has some interesting options, like the possibility of stopping or not the model checker when a given process crashes. Normally, while debugging a system, it is interesting to stop the generation of the state space if one of the processes fails. However, the existence of the option, allows the users of the tool to provoke the crashing of a given process and then study the properties under that conditions.

Another interesting option is the use of an intermediate channel introducing delays in the message sending in order to model remote semantics. In the normal way of working, sending a message is performed in an atomic way: the message is immediately placed in the queue of the receiving process. However, in a real distributed system the message can be delayed, so there is an intermediate state of the message when it has already been sent but still not received. Configuring

McErlang so that the channel is modelled this way increases significantly the state space, but it allows to check programs under more realistic assumptions.

McErlang can also be used as a simulator where the user can select which path to explore, or even as a tracing tool, executing randomly one of the possible paths.

8.5.2.2 Monitors

The monitor implements a correctness property for the *Erlang* program that should be checked. The monitor can keep an internal state if the property is complex, and it has the following interface:

```
init(Arguments) -> FirstMonState
stateChange(State,MonState, RestStack) ->
  {ok, NewMonState} || Other
```

MonState is the state of the monitor, that can be initialised or updated; State is the description of the global state with all the processes in the system; and RestStack is the structure with all the previous states of the graph. All the information in these parameters can be used by the monitor in order to check if the property holds in the state.

Other can be any value except a binary tuple with first element ok, and it indicates to the model checker that the property does not hold for the current state. A counterexample can be then generated by McErlang showing the trace back to the initial state of the graph.

8.5.2.3 Abstractions and hash tables

Abstractions are modules defined by the user that have two relevant functions: abstract_actions and abstract_state. They are called by the model checker each time a new state is generated, and allow the user of the tool to abstract multiple states into the same state.

Abstracting both actions and states is very useful in order to have smaller state spaces, but each property to be verified needs to be carefully reviewed in order to check if a given abstraction still keeps all the needed information in the state space.

A very simple abstraction, used by default in McErlang, is ordering the processes in the system state by their process identifier. The state abstractions can affect both the state of the property (monitor) and the state of the system (list of processes).

Normally states are kept in a state table by McErlang; when a new state and monitor combination is generated the tool checks whether the combination has been previously explored. If so, the new state is discarded. This is safe for safety properties; for checking liveness properties a more expressive monitor automaton is needed.

8.5.3 The McErlang approach vs etomcrl +μCRL +CADP

McErlang is still a tool under development, and therefore lacks some functionalities present in μCRL and CADP. Besides, the translation from *Erlang* to the internal McErlang language is still being developed; only a subset of the *Erlang* behaviours

are supported; moreover the memory consumption is still high, and the checker can only check for compliance of safety properties.

However, already in this stage of its development, the tool shows some advantages of the approach -model checking directly in *Erlang*- that are worth mentioning:

- The problems of using big numbers with μCRL are automatically solved. Therefore, CPU-wise, when using big numbers the graph generation is more efficient with McErlang.
- All the error messages can be easily traced back to the *Erlang* code, avoiding the limitations associated with using μCRL as intermediate language, or with the complex error messages of the tools for generating the state space.
- The translation from *Erlang* to the internal McErlang language is easier than the translation from *Erlang* to μCRL, which makes in principle the McErlang tool capable of handling a wider subset of the *Erlang* programs.
- State information is much richer than in the case of the graph generation from μCRL. In McErlang the properties can reason about any detail included in the system state (state of each of the processes, list of side-effect actions, state of the property itself), which is more powerful than using μ-Calculus over the labels of the state graph (although currently the tool is limited to check safety properties).
- The tool is very flexible and it is written in a high level programming language; as such, introducing modifications when needed for extracting a given property is always possible.
- The language of the tool is the same as the language of the developers, which makes it easier to use, or even modify it. This fits very well with the approach presented in the previous chapter of the thesis, of facilitating the task of verifying programs for Erlang developers.

Chapter 9

Extracting performance information from the *Erlang* source code

Contents

9.1 Introduction

In Part II of the thesis, the *VoDKA* server has been described in detail. In the current chapter, we will focus now the research on the distributed scheduler of *VoDKA*.

As explained in Sect. 5.2, there is a global distributed scheduler in the *VoDKA* server, meaning that the scheduling procedures and decisions are spread all over the processes composing the system, and not concentrated in one unique point in the architecture. The distributed scheduler enables that whenever a user agent is requesting a certain movie to the server, the request is transferred through the system gathering information and passing different filters and, at the end, a set of possible play-back qualities is returned back to the agent. In case of a non-empty set, the internal policy algorithms select one option and, if it is possible, the movie is streamed to the user.

The storage subsystem of the server is composed by a hierarchy of different storage systems, i.e. they could be disks, CD/DVD players or tapes. An important feature that should be remembered here is that all these devices have restrictions reflecting the available resources, of which the process controlling the device is aware. A second layer of processes controls a set of devices in one machine and has more restrictions, for example the bandwidth of its connection. A third layer may be further out in the network and serve as a cache to store more popular movies. That way the limitations and bottlenecks of the hardware resources are modelled by using the resource constraint pattern explained in Sect. 5.3.1. Again, the status of the server is distributed all over the processes that are part of the control subsystem.

Therefore, every process participating in the distributed scheduling of the system has a function determining local restrictions, given the constraints, the configuration, and the current state of the part of the system that process is aware of.

Information about the system performance is very difficult to obtain due to the distributed nature of the *VoDKA* scheduler (procedures and restrictions). For that reason, we decided to try to extract it automatically from the information we have, using formal methods. The research question is therefore: how to use formal verification techniques to extract automatically global information from the local restrictions existent in the source code and configuration of the system.

In the current chapter we explain how we have designed and used a method to construct complete models of several configurations of the *VoDKA* system. With techniques from the area of formal methods (in particular model checking) these models are used to determine global properties of the system, such as the maximum number of a certain class of movies that can be served in parallel. We are also able to extract information about the software architecture, do some tests for bottleneck detection, or even calculate the resources needed for introducing a new software component.

We are talking about performance analysis on a system directly from its source code, but there are several points of view performance analysis can be seen from. From the user point of view, we can talk about a black-box evaluation, more requirements oriented, in order to be able to measure the system capacity; the

capacity of each of the system components; or any kind of checking to see if a given scenario can be possible given a system configuration.

On the other side, from the developers point of view, performance analysis would be more focused towards architectural and internal protocol analysis, more oriented to the improvement of the design and analysis. The developers of a system such as the *VoDKA* server, are interested in tools helping them to find bottlenecks in the architecture, to extract the internal protocol and the system architecture from the source code, or to find the required capacity for a new component. After interviewing the development team at different stages of the software life cycle, those where the most demanded tools, and those where, therefore, the main goals our research was focused on.

The concrete goal of the research explained in this chapter is: given a configuration for the server (its processes, the storage devices, all the restrictions, scheduling functions, and costs), how can we extract performance information from the source code of the system?

The local properties are restrictions (on bandwidth and number of connections of disk drives, CD players, tape storage devices and such), local scheduling functions (filtering and admission policies) and cost related functions (state of the component and resources still available).

Given only these local restrictions, and the rest of the configuration of the system (number of levels and components in each level), it is far from obvious to extract information about the behaviour and performance of the system. Answering questions such as how many users can watch 'Star Wars' at the same time, is virtually impossible without building the actual configuration and testing this. Answers to such questions, however, are what both the operator of the video-on-demand server and the designers of the system are interested in. The former want to obtain information about the capacity of the system, and the later are more interested in knowing how the different distributed properties of the system influence its performance, in order to be able to know how to improve it (redesign and reconfiguration of the scheduler).

Many global properties of the system can be determined by testing, but testing all possible scenarios of users that request a movie is rather expensive. Moreover, one tests a certain configuration. Performing experiments with new drives, faster network connections and all that, increases the costs even more.

We want to be able to answer questions like the following ones:

- What is the maximum number of users in the system?
- What is the minimum number of users such that serving any MO is not possible?
- What is the minimum number of users such that serving MO1 is not possible?
- How many people can watch MO at the same time? (best case)
- How many people can watch MO1 such that the system can still serve MO2?
- How many people can watch MO1 such that serving MO2 is guaranteed?
- Would it be better to move MO from storage1 to storage2?

- Where should we move MO1 for being able to serve it to N users?
- Why (bottleneck) MO cannot be served to N users at the same time?
- What are the minimum requirements for a new component?

The idea we propose is rather general. In a design, either concurrent or distributed, where one has many processes that steer a certain functionality, one often finds global properties of the system hidden in several local properties of the running processes. Here we propose a method to reveal these global properties by a kind of exhaustive simulation of the system.

9.2 Method

In order to extract global properties from an *Erlang* system, we build a model of the system by using a set of tools in a way described in this section. We use the tools in a rather unconventional way, and we show how the obtained model can be used in a concrete case study described in the next section.

Our goal in the development of the methodology can be summarized as: given a real distributed system, defined by its functional requirements, the design and implementation of the system, and its performance requirements; using techniques from the formal methods field, composed by a compiler from *Erlang* to process algebra, some process algebra tools, and some model checking and graph analysis algorithms; we want to find and fix functional problems, performance problems and design problems (related to the system maintainability and its flexibility). This briefly defines our input, the tools we use and the output we want to achieve. In the next paragraphs all these ideas are explained in detail.

First, we translate *Erlang* to a μ`CRL` specification with the `etomcrl` tool described in Sect. 8.2. Second, we use μ`CRL` and the `instantiator` explained in Sect. 8.3 to generate the state space of our μ`CRL` specification, and therewith of the *Erlang* code. The state space is then reduced using `CADP` in order to hide all the information that is not relevant for extracting the performance information.

In the reduced graph the failures of requests are visible and therefore, the shortest path to a failure. This answers the question on how many users are guaranteed to be able to be served in parallel. Other questions, such as 'How many people can watch the movie A such that the system can still serve B?' or 'Where should we store the movie A for being able to serve it to N users?', can also be expressed as properties of the graph.

We designed a user interface to guide the whole process: choosing the parameters of the configuration, generating the model, constructing the graph and translating human understandable global properties of the system into a temporal logic formula. This formula is checked by model checking techniques using `CADP`, already introduced in Sect. 8.4. The formula is used as a declarative way of asking for knowledge of the system and the checking techniques primarily as efficient graph search techniques.

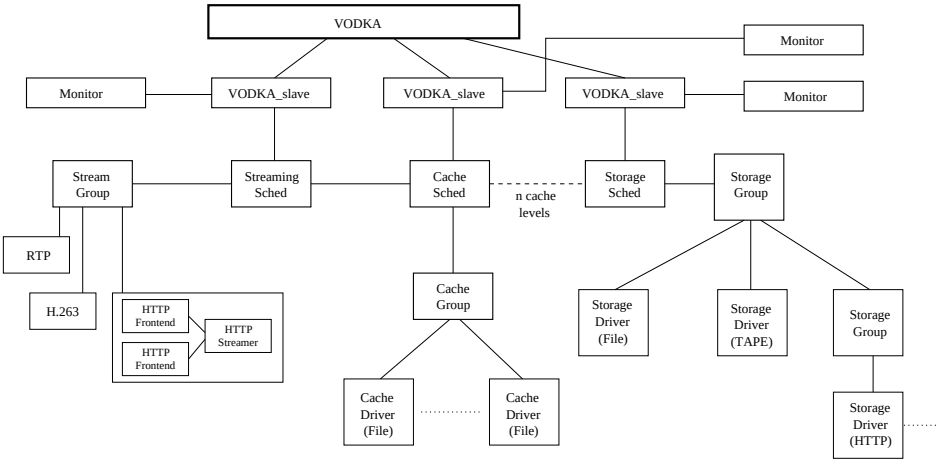


Figure 9.1: Architecture configuration example for the VoD system

Case study: the *VoDKA* video on demand server Scheduler

Here we present the concrete architecture used in our case study. Although the details of the *VoDKA* server have been already described in the previous part of the thesis, we include a very short explanation of the most relevant information needed for a better understanding of the current chapter. In concrete, we briefly describe the distributed scheduler and propose an example of an architecture configuration that is going to be taken as the case study. The case study is itself a simplified version of *VoDKA* where some of the details that are not relevant for our research have been hidden.

The system flexible architecture is based on a hierarchy of specialized levels that can be combined in different ways, depending on the needs for a given deployment of the server. A common configuration of the system architecture would be: a massive *storage level*; one or more *cache levels*, that are going to reduce the performance requirements (i.e. the response time or the bandwidth) of the lower levels; and a *streaming level*, that implements the protocol adaptation between the server and the user client. The software of the system has been developed using *Erlang/OTP*, and it is deployed over an architecture of *GNU/Linux* based clusters of commodity computers.

In Fig. 9.1 the general architecture of the system for a linear configuration is shown. The boxes in this figure correspond to an *Erlang* process. Each of the levels is composed by a set of software components (most of them are *Erlang gen_server* processes) with a standard API. On top of the generic server based architecture, a supervision tree is constructed for providing fault tolerance. The system can roughly be divided in three levels: *storage level*, *cache level* and *streaming level*. Any *storage level* is composed by a *storage scheduler* and a hierarchy of storage devices grouped by one or more *storage groups*. The storage level is connected to the *streaming level*, but one or more *cache levels* may be in between them. The structure of the cache level is similar to that of a storage level; only logically they differ, since *Media Objects* are dynamically copied to it and removed after use. The *streaming level* has a hierarchy of components implementing the adaptation

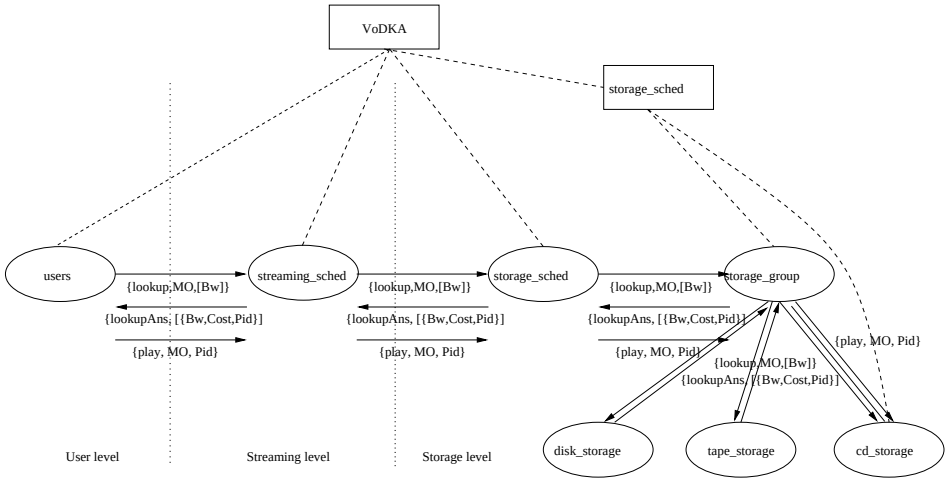


Figure 9.2: Detailed configuration example without cache level

to the streaming protocols accepted by the system (HTTP, RTP, H.263, etc.). The processes in the streaming level create and supervise all the processes needed for performing the actual transmission of data through the system.

All the processes of the system have local restrictions, cost functions and decision algorithms, and all these *configuration* values are going to determine the distributed scheduling of the multimedia server.

Whenever a user requests a given MO with a concrete quality (bandwidth), this request is received in the streaming level and propagated trough all the levels of the system. If the MO can be provided by a given level, because the object is stored there and it has the resources left to provide it, then this information is returned. The scheduler is in charge of elaborating a list of candidate providers, with an associated *cost*, that is going to be sent back to the upper levels of the hierarchy. Finally, after filtering the options in the different levels on the way back to the user, either a *fail* or the opportunity to *play* the MO is replied.

In this chapter, a method is described to automatically derive the global scheduling performance properties of the system from the local restrictions on the scheduling subsystems. We concentrate on configurations of *VoDKA* in which we have one streaming level and one storage level without any cache level. One may argue that a cache level behaves like a storage level in our performance analysis, since we consider in the storage level all possible distributions of the *Media Objects* over the devices (i.e., also copies of the same object on multiple devices). For an average-case scenario, the dynamic behaviour of the cache may become interesting, but for a worst-case scenario, where all users start asking for a set of movies at the same time, the cache can be seen as static storage. Fig. 9.2, the concrete group of processes used for the case study are shown. Squares represent supervision processes, and spheres worker processes (worker in *Erlang* terminology is the name given to the processes carrying out the actual work in a system). The dotted lines represent supervision links, and the arrows are example messages that are exchanged between the processes. This is an abstraction of the real software, because we have decided to hide some of the low level processes, that are not relevant for the analysis.

In summary, the *VoDKA* server has a completely distributed scheduling subsystem, without any kind of global state or global decision mechanism. The scheduling algorithm is distributed among all the different processes in the software architecture. Each process in the scheduling subsystem can implement: restrictions (number of connections, maximum bandwidth), scheduling function (filtering, cache algorithms, admission policy), and cost (state of the component and resources still available).

In the following sections, we will use mainly three configurations of the explained the case study. All of them follow the architecture explained above, with one streaming level, no cache, and a group of storage devices. The configurations are the following ones:

- **Configuration 1 of *VoDKA*** :
  - Connection of the streaming level with the storage level:
    - Maximum bandwidth: 15 units.
    - Maximum number of connections: 20.
  - Connection of the storage level with the devices:
    - Maximum bandwidth: 15 units.
    - Maximum number of connections: 15.
  - List of devices and restrictions in each device:
    - Tape storage with:
      - 1 maximum connection.
      - 2 units of bandwidth.
      - Movies of type $m1$ and $m3$. Both types are available with bandwidth rate 1 and 2.
    - CD storage with:
      - 2 maximum connections.
      - 3 units of bandwidth.
      - Movies of type $m1$ and $m2$. Both types are available with bandwidth rate 1 and 2.
- **Configuration 2 of *VoDKA*** :
  - Connection of the streaming level with the storage level:
    - Maximum bandwidth: 15 units.
    - Maximum number of connections: 20.
  - Connection of the storage level with the devices:
    - Maximum bandwidth: 15 units.
    - Maximum number of connections: 15.
  - List of devices and restrictions in each device:
    - Disk storage with:
      - 4 maximum connections.
      - 4 units of bandwidth.

            - Movies of type $m1$, $m4$, $m6$ and $m7$. All of them are available with bandwidth rate 1 and 2.
        - Tape storage with:
            - 1 maximum connection.
            - 2 units of bandwidth.
            - Movies of type $m3$, $m5$, $m6$ and $m7$. All of them are available with bandwidth rate 1 and 2.
        - CD storage with:
            - 2 maximum connections.
            - 3 units of bandwidth.
            - Movies of type $m2$, $m4$, $m5$ and $m7$. All of them are available with bandwidth rate 1 and 2.

- **Configuration 3 of *VoDKA*** :
    - Connection of the streaming level with the storage level:
        - Maximum bandwidth: 14 units.
        - Maximum number of connections: 20.
    - Connection of the storage level with the devices:
        - Maximum bandwidth: 16 units.
        - Maximum number of connections: 30.
    - List of devices and restrictions in each device:
        - Disk storage with:
            - 5 maximum connections.
            - 6 units of bandwidth.
            - Movies of type $m1$, $m5$, $m6$, $m7$, $m11$, $m12$, and $m15$. All of them are available with bandwidth rate 1 and 2.
        - Disk storage with:
            - 4 maximum connections.
            - 4 units of bandwidth.
            - Movies of type $m2$, $m5$, $m8$, $m9$, $m11$, $m12$, $m14$ and $m15$. All of them are available with bandwidth rate 1 and 2.
        - Tape storage with:
            - 1 maximum connection.
            - 2 units of bandwidth.
            - Movies of type $m3$, $m6$, $m8$, $m10$, $m11$, $m13$, $m14$ and $m15$. All of them are available with bandwidth rate 1 and 2.
        - CD storage with:
            - 2 maximum connections.
            - 3 units of bandwidth.
            - Movies of type $m4$, $m7$, $m9$, $m10$, $m12$, $m13$, $m14$ and $m15$. All of them are available with bandwidth rate 1 and 2.

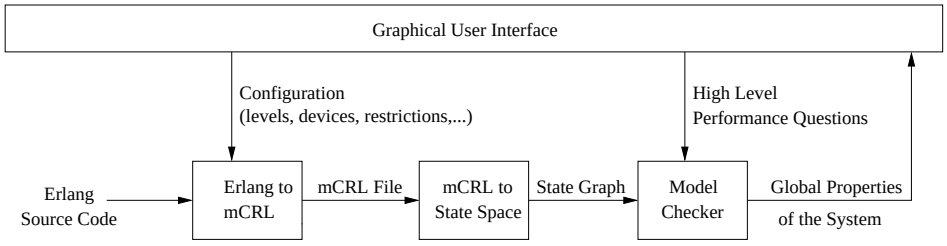


Figure 9.3: Proposed three steps methodology: from *Erlang* to global properties

In all the configurations, we will have an unbounded number of users asking randomly for all the *Media Objects* with all the available bandwidths. This will provoke all possible scenarios inside the system to be inspected, depending on the order the requests are sent.

There is other important configuration value: the cost. The cost of each component of the system allows the internal filters and scheduling algorithms to select the component from where the content is going to be streamed. In our case, we have selected realistic values based on the configuration of real system deployments.

Going from local restrictions to global properties

The proposed methodology consists in generating the full state space of the system from its configuration, starting directly from the *Erlang* source code, which is easier if we use the knowledge about the *Erlang* design patterns (behaviours). From the system state space we hide the information that is not relevant, and we obtain a new reduced graph, where we can extract the information we are interested in, by model checking. The schema of the method is shown in Fig. 9.3.

The input source code is already an abstraction of the real one, where only the scheduling subsystem is taken into account, and no resources are released. So we construct in *Erlang* model we want to analyze by abstracting from the low level details present in the actual implementation.

μ`CRL` process algebra is used as intermediate step, because it has efficient tools for generating the state space, and its semantics are quite similar to *Erlang*.

A high level GUI separates the theoretical details from the users of the methodology. In the case-study at hand, we are interested in the performance of several configurations of the video server, without actually building all possible configurations. Moreover, we would like to obtain some insight in the system, such that we get an idea on how to improve the software.

As we have explained in Sect. 8.2.2, the full state space of a system consists of the combination of all possible states that can occur in the state parameter of a call-back function. This is a real reduction and in many practical examples it is a finite state space.

The events in the system are the messages the particular servers receive and reply/send. In our example, the messages are requests for *Media Objects* that are passed from one level to the other, and a list of choices propagated in return. As long as the list of *Media Objects* is finite, this results in a finitely branching graph. However, in case of a realistic number of movies, this would be an enormous graph,

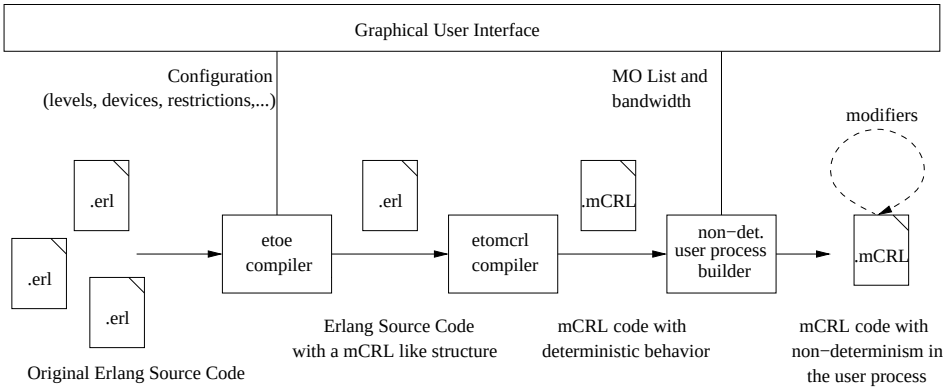


Figure 9.4: `etomcrl`: from *Erlang* to μ`CRL`

impractical for our purposes. We realized that the level of detail on the specific movie is unimportant for the analysis of the system. What is important is that it is a *Media Object* that is stored on the first disk, or that it is an object stored on both disk and tape. Hence, we look at configurations of the system in which we instantiate our storage devices with abstract objects $m1$, $m2$, *etc.* Typically, we have one object per combination of devices. Thus, there is one object that is both on tape and on disk; one object only on tape, one only on disk, *etc.* The real distribution of movies is a function from *Media Objects* to abstract *Media Objects*. The user of our tools can therefore still ask the question: "Is it possible to have 30 users watch 'Star Wars' at the same time?". The abstract object is computed and the question translated.

As a real advantage of these abstract *Media Objects*, we get for free that we can answer the user: "No this is not possible, but if you put one extra copy on CD player 4, then this is possible". We just try to answer the question for all possible abstract *Media Objects* and determine the difference between the given abstract object and the abstract object for which it succeeds.

The analysis tool consists of three parts that are connected by a graphical user interface (GUI). First, the user interface lets the user select a configuration, i.e. the levels in the system, the storage devices with their limitations and the *Media Objects* stored on each device. The interface then calls our translation tool, that translates the system with this particular configuration to a μ`CRL` specification (Sect. 9.2.1). Second, the user interface activates tools to efficiently generate the state space and reducing it (Sect. 9.2.2). Third, several properties are presented in the user interface that can be checked automatically using the underlying model checker and some gluing software (Sect. 9.2.3).

9.2.1 Step one: *Erlang* to μ`CRL`

In Fig. 9.4 the translation from *Erlang* to μ`CRL` is sketched. The first two steps are performed with the `etomcrl` compiler. The last step is a small modification to the created μ`CRL` code to obtain a non-deterministic *user process*.

The `etomcrl` *Erlang* to μ`CRL` compiler is meant for systems that consist of servers that are implemented using the generic server design pattern and clients

(that might be servers as well) that are restricted to communicate via the generic server API (i.e., *gen_server:call* etc.). A set of *Erlang* modules implementing that servers and clients forms the input of our compiler. In *Erlang* the processes implemented by these modules can be dynamically created and normally a supervision tree will be used to initiate that. Our case study fits perfectly in this kind of systems.

The first task for the compiler is to symbolically evaluate the supervision tree for a given set of arguments (determining the configuration) to find the processes and their initial arguments that are used in the system. After computing the set of processes in the system, every module implementing one of these processes is translated. These processes are either simple clients or arbitrary complex generic servers. The gap between *Erlang* and μCRL is solved by following all the translation steps described in Sect. 8.2.

The `etomcrl` tool is used in our case study, therefore, in a very automatic and transparent way. We execute it with the same arguments than the function that would start the *Erlang* supervision tree, but instead of that, the code is translated and a μCRL file.

The obtained μCRL specification can be used to generate the state space of the *Erlang* system. However, in our scheduler we are not ready yet. As our main goal is to automatically obtain global properties about the performance and behaviour of the system, we only have to analyze the control core of the video-on-demand server. But for 'activating' the system, a special *interface* that represents the possible users asking for *Media Objects* has to be added to the model. One direct solution would be to add to the *Erlang* source code an abstraction of the user process and to include this in the supervision structure of the system, thus introducing the number of users as a parameter in the same level as the number and configuration of the different devices. The problem of this approach is that it would be hard to explore all the different combinations of users, because the fact of including a lot of new processes would make the state space to grow exponentially. In the state space with many users one would get different paths for different orders in which the users ask for *Media Objects*. We are, however, not at all interested in the difference between user one asking for 'Star Wars' followed by user two asking for 'Star Wars' and the sequence in which they ask it the other way around. The only thing that is important for us is that two users asked for 'Star Wars' after each other. Another disadvantage with the solution of many user processes is that it is hard to tell how many of them will be needed to determine the capacity of the system.

The solution we choose lays in a different approach for modelling the users: in the *Erlang* source code, only one client process is used to model the *user pattern*. That process asks for a *Media Object* with a given quality, waits for the answer (the set of options provided by the server), and plays the object or asks again (if the request fails), repeating this loop all the time. Then, after creating the μCRL file, we use our tool for automatically adding non-deterministic behaviour to the user process. Instead of asking for a concrete movie with a given quality, now the user is going to ask non-deterministically for one out of all possible *Media Objects* with one out of all the possible qualities. Thus, we use one non-deterministic user process for modelling an infinite set of users that constantly request *Media Objects*. Since the user process never releases a *Media Object*, this process will only receive

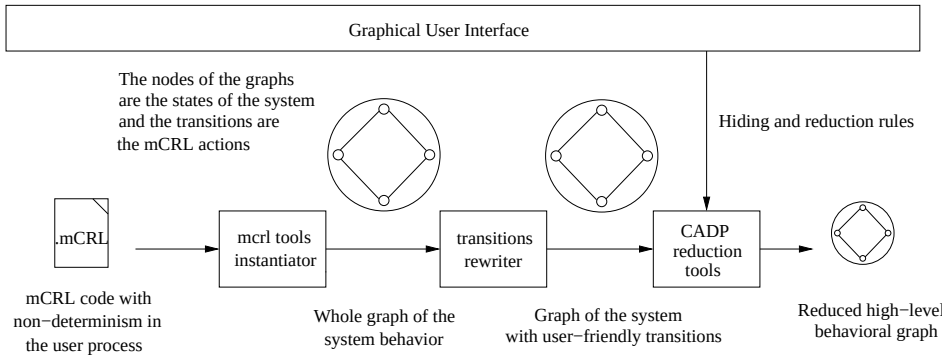


Figure 9.5: From μCRL to state space of the system

'fail' as a reply after that the system is overloaded. As such, there is a natural bound on the number of users in the system and the creation of an infinite state space is avoided.

The tool set for μCRL comes with several other tools that can be used to modify the μCRL specification before generating the state space [GL01b]. The aim of these modifications is to end up with a specification that gives less states and transitions (events) in the generated state space. This can theoretically be achieved because some events, like the communication with the *call-stack* or the *buffer* may be seen as internal actions and can be hidden in the state space. Tools like a confluence analyzer can be used to modify the source code in such a way that only one of the many confluent paths to a result is chosen in the generation of the state space, provided that the obtained state space is observational bisimular with the original one. We experimented with these tools as well, giving reductions of ten to twenty percent in the generated state spaces.

9.2.2 Step two: Generating a State Space from μCRL

After generating the μCRL code for the concrete configuration of our system, the second step of the proposed methodology is to create the state space for that configuration. This step is based on standard tools for μCRL [SEN99], and tools for hiding and renaming labels as well as the reduction tools in the Cæsar/Aldébaran tool set [FGKM96]. Fig. 9.5 depicts which tools are used to create a reduced state space from the process algebra specification.

In the *Erlang* scheduler software there are many processes that communicate with each other. The communication is rather straight-forward, though. The user sends a request, this is passed from one process to the other and in the end a growing list of possibilities is passed back.

For the typical properties we are interested in, we are only concerned about the messages that the user sends to the system and the messages that are returned to the user. In that way, we can judge whether a user can be served and what the possible choice for the user are. The messages between streaming level and storage level, between storage group and devices, are irrelevant for our purpose. However, the translation from *Erlang* to μCRL is such that when we use the *instantiator* tool on the specification directly, we obtain a state space in which all these

irrelevant messages are visible as well. Therefore, the state space contains at least $4 * (2+\#\text{devices})$ times more events than we are interested in, whereas in practice this redundancy turns out to be even larger.

We generate the full state space and then rename the labels of the events we are interested in and we hide the other events. By using reduction tools we can perform observational bisimulation reduction on the state space, obtaining in this way a much smaller state space in which only the relevant events are shown.

Renaming the labels is useful to create a better readable visualization of the graph. For small configurations such visualizations can be illustrative for the designers to look at. Moreover, the properties that we use later refer to the renamed labels. If we would not rename the labels, the properties would be harder to formulate. Hiding the irrelevant events has the two advantages that the properties can be formulated configuration and system implementation independent (they need not reflect internal behaviour) and that model checking them is faster.

Fig. 9.6 shows a very small illustrative example of a reduced state space obtained from a simple linear configuration. The system was configured with two linear levels (streaming level -with the local scheduler-, and massive storage level -with the scheduler and the storage group). The storage group was grouping two devices: a tape with 2 units of bandwidth, only able to handle one connection at the same time; and a CD with 3 units of bandwidth able to handle 2 simultaneous connections. No extra restrictions were placed in the hierarchy, other than trivial cost functions able to select the right providers for the users. In this example, we used the abstract approach for the *Media Objects*, placing in the devices all the possible combinations of MOs. That is, to have *m1* as the abstract kind of MOs that are in both devices, and *m2* and *m3* as the MOs that are only in one of them. For the quality of the MOs we chose to have only two possible qualities with 1 and 2 units of bandwidth respectively.

For the example configuration, from the original whole state space of the system, with a total of 2547 states and 2747 transitions, the reduction results in the 8 states and 48 transitions shown in Fig. 9.6. In the graph, the performance pattern for this kind of systems can be seen; from the initial state 0, the transitions explain which are the actions the users are able to perform (e.g. *play(tape,m3,1)* means to serve an MO from the group *m3* on the device called *tape* using 1 unit of bandwidth). After more users requesting movies, the system is becoming more busy. Some resources are not available to handle some a user request (e.g. in state 5 the system is not able to provide the user an MO from group *m2* at a quality of 2 units bandwidth, because that media is only on the CD device, and the device only has 1 unit of bandwidth available at that moment). The last node of the graph is always representing the maximum load of the system, where all the resources are being used (the bottlenecks of the architecture are using their maximum capacity), and every user request results in a *fail* as reply.

9.2.3 Step three: Performance analysis with model checking

9.2.3.1 Verifying Global Properties

Once we have generated the reduced state space, representing the behaviour of the system from a black box point of view, the last step of the proposed methodology

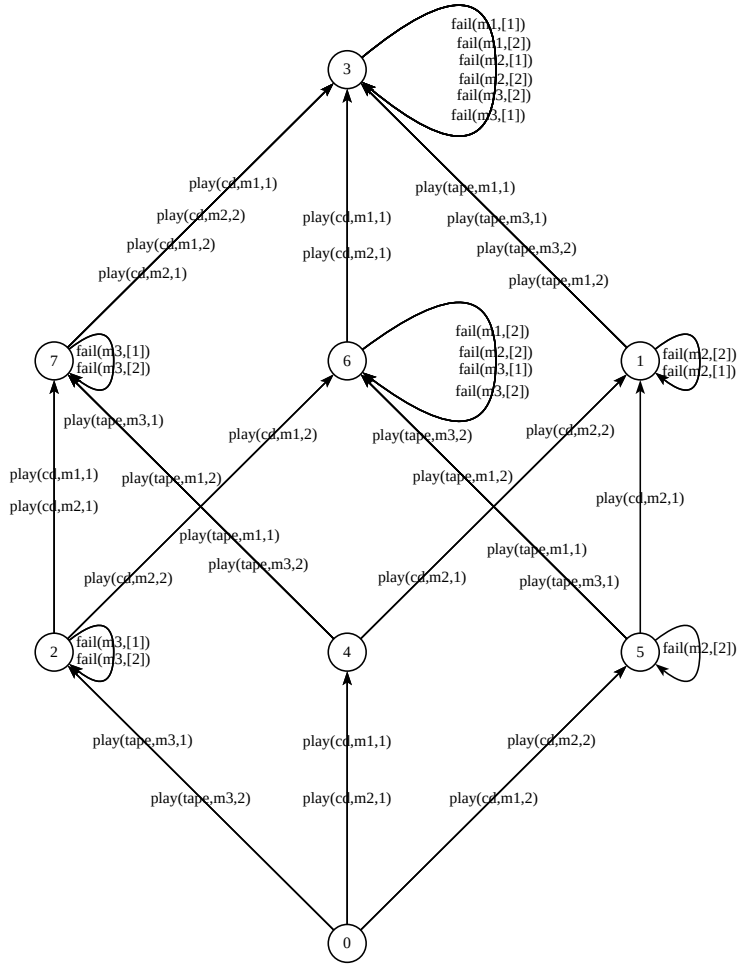


Figure 9.6: Abstract graph for a simple configuration

is to extract the performance properties from this graph by using model checking techniques.

An important goal in this step is to provide the users of our tool (both the designers trying to improve the performance and the cable operator evaluating the system) with an understandable set of properties presented in natural language via a graphical. user interface. The user is given the possibility to ask questions like "How many movies can the system serve simultaneously?" instead of "What is the longest path in the state space to the point where only *fail* transitions are possible?". Another desirable feature when designing this part of the methodology is to provide the user with some kind of feedback information giving design related suggestions.

One of the main subsystems of the GUI developed in our prototype is devoted to this high level interface. The user can automatically check some properties, and can formulate others in a very easy way, giving, for example, a description of a concrete branching scenario in order to know if that can happen in a real execution of the system. Internally, these properties are converted to an alternation free μ-`Calculus` expression, that is going to be model checked using the `CADP` tools; therefore, the users of our tools and method do not need to deal directly with the logic.

The properties we offer the user to be analyzed are divided in three main groups, depending on the way the model checking techniques are used in order to obtain the information from the graph:

1. Counter-example based:

   With this method for extracting system information we propose a new way of using model checking tools. Instead of checking the property in order to know if it holds for the graph we are analyzing, we try to find the negation of the formula that represents the information we want to obtain. Therefore, a counter-example to the formula corresponds to an example that the global property holds, thus the counter-example gives us the knowledge about the system.

   One of the main properties a user would like to know about this kind of systems is its global capacity, but the abstract concept of *capacity* is really measured by a complex set of properties. One of them would be A way of obtaining that information, using the counter-example based approach, is to use a property that we know is only false in the special node of the system where only fails can take place, viz. `[true*]<not 'fail.*'>true` (i.e. starting at any state of the system, it is always possible to have a transition that is not a *fail*). The counter-example for that property is going to be (with the tools we are using) one of the shortest paths to that node.

   Other properties for the systems, related to its performance and capacity, are obtained using the same approach. Instead of looking at the whole capacity of the system, we can focus on a concrete *Media Object*, and know the server performance for that *Media Object*. By using the μ-`Calculus` property `[true*]<not 'fail(.*,m1,.*)'>true`, the tool can automatically obtain a counter-example with the shortest path to a fail for that *Media Object*. Some

other similar expressions are used in order to check the worst case performance for the system where only plays are possible, e.g. the shortest path to any kind of fail (giving us a capacity idea about for how long the system is still able to always serve the user).

Another interesting property is the maximum of *Media Objects* that can be served simultaneously in such a way that after serving these objects all possible requests can still be honored. Thus, the paths where all states only have successful successor states. In our little example in Fig. 9.6 this corresponds to the path from 0 to 4. Thus, in that example, the maximum number of simultaneous users after which a next requesting user always can be served is therefore 'one'. In general one can have several distinct paths in which all users can still be served. With this counter-example based approach we automatically determine the shortest of them by using the μ-`Calculus` formula:
`[true*](<'fail.*'>true<true>['fail.*']false`.
The longest of them can be detected using the logarithmic search approach described below.

Thus, with the proposed technique the user can obtain automatically, by using a high level user interface, scenarios that fulfill global properties of the system behaviour. All these properties can be combined with the bandwidth usage in order to extract more detailed information.

2. Logarithmic search:

   Complementing the counter-example based approach, a different set of global properties of the system can be obtained by using model checking in combination with a fast search algorithm with logarithmic complexity. With the counter-example approach some interesting results cannot be extracted from the graph, because of the limitation of the `CADP` model checker tools, that are giving always a shortest path to a node where the property does not hold.

   One kind of properties the users of our system are interested in are reflected by questions like: "What is the maximum number of users that can watch 'Star Wars' at the same time?" or "How many simultaneous users can the system provide such that it still is always able to play 'Star Wars'?". Expressed in a property over the graph these questions refer to the longest paths to some special nodes or situations.

   As an illustrative example we have the previously mentioned maximum number of simultaneous users, such that the next requesting user can always be served. The longest path to a state where all requests are successful is found by repeatedly proving properties of the form:
   `<true*.'play.*'.true*...>['fail.*']false`, where the length of the sequence of the second `true*` occurrence is varied doubled until the property is false and then we search the exact point of failure by taking the middle between previous success and previous failure recursively.

   These properties, extracted also from the graph but using a different approach, complement the information obtained with the counter-examples for giving the user global properties of the system. These properties are again

obtained automatically, with only high level user interaction through the GUI.

3. Scenario based:

   Finally, as the third kind, the tool also provides to the user a more open interface for expressing scenario-like properties in almost natural language, that later are transformed internally into μ-`Calculus` expressions. Examples of this kind of properties are the *existential* properties, where the user can describe a concrete scenario (with *Media Objects* and bandwidths) and ask the system if that can happen; and the *eventually existential* properties, where the user can describe a more complex scenario in the form 'after looking at a given sequence of *Media Objects* with a given quality, it is still possible to have the following scenario: ...?'.

   With these kind of properties, the abstraction of the *Media Objects* placed in the system can effectively be used. If the user asks the tool whether a given scenario is possible, the tool checks this scenario for all abstract movies, instead of just the one that the user asked for.

   For example if the users ask whether the system can provide the following sequence of movies: movie 1, movie 3, and movie 2. Assuming that movie 1 and 3 are both belonging to group $m1$ and that movie m2 belongs to group $m3$, then to answer *yes* or *no* to this question, the following property is checked:

   ```
   <'play(.*,m1,.*)'.'play(.*,m1,.*)','play(.*,m3,.*)'>true
   ```

   However, if the answer to the question is *no*, then we can also automatically check the property where we smartly exchange $m1$ and $m3$ by other groups. If one of these properties is true for the state space, we obtain a result that can be presented as an advice to the user. In this way, the tool can return a more complete answer like: "the scenario cannot occur, but moving movie 2 from the tape to the second CD makes this scenario possible in the system".

Thus, by combining three (complementary) ways of using of model checking techniques a method is given for automatically verifying global properties of the system.

9.2.3.2 Architecture from the messages

One interesting option once we have the ability to create the full state space for the system is to try to extract the software architecture (this meaning all the processes and how they communicate with each other) automatically from the state space. In order to do this, we use the extension to `etomcrl` that was introduced in Sect. 8.2.5.

Analyzing the *Erlang* generic servers and other behaviours, we can see that information about source processes and destination processes, together with the content of the messages, are explicit information in the process algebra actions, and can be easy, using our framework, extracted from the code. Doing this, we are

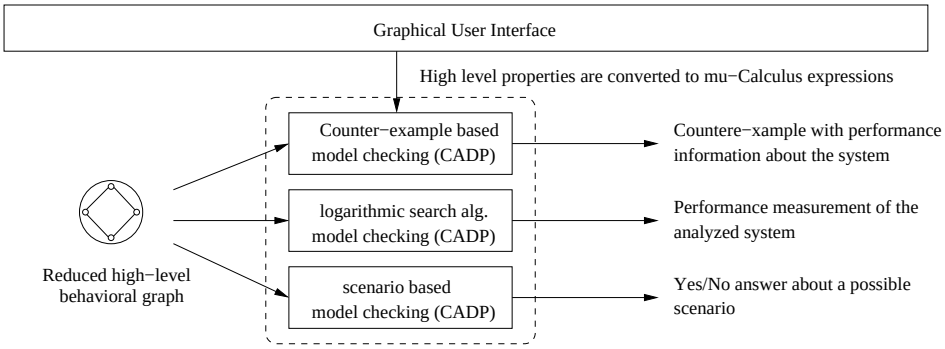
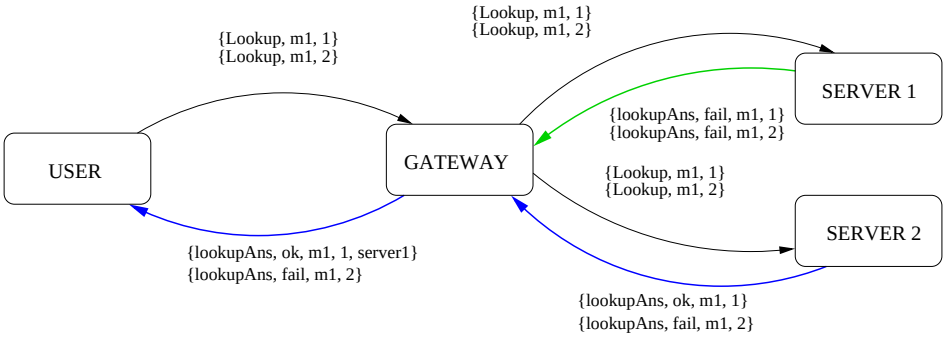


Figure 9.7: From the behavioural graph to the global performance properties



able to build graphs similar to the one in Fig. 9.2.3.2, where the boxes represent processes and the arrows the messages they exchange.

This kind of information extraction has shown to be very useful during the different stages of the system development, as a way of improving the system understanding. In some cases, we found errors in the *Erlang* implementation, because the communication was taking place between the wrong processes. In other examples, the graph was useful for increasing the understanding of the information we were extracting from the system.

9.2.3.3 Bottleneck information

A bottleneck is a point inside the system architecture that in some way is reducing the whole system performance, being the first part of it which collapses at a given system load. This can be seen as a quite generic idea, which means that several more concrete definitions can be distinguished. In the case of the *VoDKA* server, three main bottleneck definitions could be found:

- Internal independent bottleneck: *the first place were we can see a fail in the system, in any of the possible execution paths.* They can be extracted automatically stopping the graph generation when any kind of fail occurs in the system. In order to do that, we can use the extension to `etomcrl` described in Sect. 8.2.4. All we need to do is determine which messages are showing a fail in the system, annotate the modules where we want to

find bottlenecks, and run the tool. The advantage of stopping the graph generation when a bottleneck appears is that we can handle potentially much bigger state spaces.

- External/user independent bottleneck: *the point in the architecture that makes the first fails to be answered to a user request.* They can be automatically obtained from the state graph by analyzing the fails in the top level. We could either work with the reduced graph, generating first the complete one and then reducing; or using again the extension for observing fail messages in order to stop the generation. We would normally use the second option for really big state spaces.

- Internal relative bottleneck: *the part of the system where a fail in a component is too far away from a fail in a different component.* We can extract this kind of information by model checking the graph with formulae talking about the distance between fails. An example of this would be to state that after a failure in component A, we should not be able to see X successful messages before seeing a fail in a different component. This idea of bottleneck is assuming that the systems work better if they are balanced, i.e., that they reach their maximum capacity almost at the same time under stress conditions.

For all of them, using graph analysis tools, we can extract a table summary with information about the bottlenecks in the system.

9.2.3.4 Calculating resources for a new component

In the previous sections we have seen different approaches inside our method in order to extract information about a given configuration of the system. But an interesting question from the designers point of view would be: "if I am going to add a new component to the system, for example a new storage device, how many resources (storage capacity, communication bandwidth) should I include in the resource so that the overall system performance is going to be X?" (being X, for example, the system performance given in terms of number of movies that can be seen at the same time). In other words: how many resources we need to provide a new component with in order to avoid it to be a bottleneck point in the system architecture.

We can use the same a similar approach than the previous ones: adding to the system architecture the new component without any resource restriction. We compute the system capacity in the execution graph, and we extract by graph analysis the information about the maximum number of times that the new component is asked. The new component can be designed (and its restrictions adapted) in order to be able to serve all the possible requests that is going to receive, thus avoiding it to be the bottleneck.

In case the amount of resources needed is impossible to satisfy, or too expensive, we still get a lot of useful knowledge from the system analysis and can use it in order to reorganize the system configuration for satisfying the new requirements.

9.3 Results

In this section we will analyze the results of the method presented, applied to the *VoDKA* case study. In concrete, we will discuss the results of applying them to the three configuration examples that were described. In some cases, for illustration purposes, other experiments will be used. The goal of the examples is neither to explain the exact limitations of the tool nor to show a comprehensive set of experiments carried out. All kind of configurations, including more complex examples have been tried during the development of the thesis, but only those three are discussed here for clarity purposes.

9.3.1 Intermediate results of the experiment

9.3.1.1 μCRL model generation

The translation using `etomcrl` worked perfectly in all the examples we tried. The time it takes in all the cases is a matter of milliseconds[1], so it is not relevant in the whole process (the bottleneck is in the state space generation, not here).

The three *VoDKA* configurations described use the same *Erlang* model, so the translation is similar. In fact, the configuration is only reflected in the initialization of the processes and the code of the user process.

A simplified version of the `init` part of the μCRL specification for *VoDKA* in its `Configuration 1` would look like:

```
CallStack(empty) ||
cd_storage_init(cd,cons(cons(tuple(m1, tuplenil(cons(int(s(0)),
  cons(int(s(s(0))),nil)))),cons(tuple(m2,
  tuplenil(cons(int(s(0)),cons(int(s(s(0))),nil)))),nil)),
  cons(int(s(s(0))),cons(int(s(s(s(0)))),cons(int(s(0)),
  nil))))) ||
Server_Buffer(cd,emptybuffer) ||

CallStack(empty) ||
tape_storage_init(tape,cons(cons(tuple(m1, tuplenil(cons(int(s(0)),
  cons(int(s(s(0))),nil)))),cons(tuple(m3, tuplenil(
  cons(int(s(0)),cons(int(s(s(0))),nil)))),nil)),
  cons(int(s(0)),cons(int(s(s(0))),cons(int(s(0)),
  nil))))) ||
Server_Buffer(tape,emptybuffer) ||

CallStack(empty) ||
storage_group_init(storage_group,cons(tape,cons(cd,nil))) ||
Server_Buffer(storage_group,emptybuffer) ||

CallStack(empty) ||
storage_sched_init(storage_sched,storage_group) ||
Server_Buffer(storage_sched,emptybuffer) ||

CallStack(empty) ||
```

[1]For running all the experiments we present here we have used an IBM Thinkpad R51 with a Pentium Mobile 1,5MHz CPU and 512 MB of RAM

```
streaming_sched_init(streaming_sched,storage_sched) ||
Server_Buffer(streaming_sched,emptybuffer) ||

users_init(pid(0),streaming_sched)
```

In the code above, all the hiding and encapsulation that is automatically done in the μCRL level by the `etomcrl` tool has been removed for clarity reasons. For each *Erlang* process, three processes are started: the call-back, the buffer, and the process implementing the logic. The arguments specify the restrictions and the available movies in each of the devices, together with the identifier of the processes they need to communicate with.

After generating the μCRL translation, the `users` process is changed as explained in the method, giving place, in case of `Configuration 1` to the following code:

```
users_loop(MCRLSelf:Term,StreamingScheduler:Term) =
 (gen_server_call(StreamingScheduler,tuple(lookup,
      tuple(m2, tuplenil(cons(int(s(0)),nil)))),MCRLSelf) +
  gen_server_call(StreamingScheduler,tuple(lookup,
      tuple(m2, tuplenil(cons(int(s(s(0))),nil)))),MCRLSelf) +
  gen_server_call(StreamingScheduler,tuple(lookup,
      tuple(m1, tuplenil(cons(int(s(0)),nil)))),MCRLSelf) +
  gen_server_call(StreamingScheduler,tuple(lookup,
      tuple(m1, tuplenil(cons(int(s(s(0))),nil)))),MCRLSelf) +
  gen_server_call(StreamingScheduler,tuple(lookup,
      tuple(m3, tuplenil(cons(int(s(0)),nil)))),MCRLSelf) +
  gen_server_call(StreamingScheduler,tuple(lookup,
      tuple(m3, tuplenil(cons(int(s(s(0))),nil)))),MCRLSelf)).
  ...
  users_loop(MCRLSelf, StreamingScheduler)
```

The user asks randomly for the three kinds of movies present in the system with all the possible bandwidths.

9.3.1.2 State space generation

The generation of the state space from the μCRL specification is done for all the examples in a very automatic way, following the steps explained in the method. In the case of *VoDKA*, after generating the global state space, we use the SVL script language included in `CADP` in order to renaming the labels (getting a more *Erlang* syntax that makes easier to trace analysis back to the original source code), hiding all the internal information not interesting for measuring the system capacity, and obtaining an equivalent reduced graph.

The obtained results are very different in size and computation time for the three configurations:

- `Configuration 1`:
  - Size of the original state space: explored 2547 states, generated 2747 transitions, and 137 levels.
  - Generation time for the original state space: user 0m7.744s / sys 0m0.192s.

  - Size of the reduced state space: 8 states, 48 transitions.
  - Reduction time for the state space: user 0m2.520s / sys 0m0.636s.

- `Configuration 2`
  - Size of the original state space: explored 62608 states, generated 69888 transitions, and 321 levels. The size of the AUT file generated is a bit less than 4 MB.
  - Generation time for the original state space: user 0m55.135s / sys 0m0.696s.
  - Size of the reduced state space: 40 states, 560 transitions.
  - Reduction time for the state space: user 0m3.464s / sys 0m0.764s.

- `Configuration 3`:
  - Size of the original state space: explored 2299914 states, generated 2655048 transitions, and 583 levels. The size of the AUT file generated is 157 MB.
  - Generation time for the original state space: user 44m56.525s / sys 0m26.346s.
  - Size of the reduced state space: 512 states, 15360 transitions.
  - Reduction time for the state space: user 0m52.775s / sys 0m1.516s.

As it can be seen from the numbers above, for small examples, the transformation from the *Erlang* code and the concrete configuration to the abstract behavioural state space is performed in a matter of seconds. As we complicate the system configuration, the state space grows and the computation time is much larger.

The main performance limitations of the approach is the time it takes the state space generation tool of the *μCRL tool set*. As we said, our tool is able to handle reasonable sized real configurations, but when the goal is to analyze a really complex tree-like hierarchical architecture, with a big number of devices and restrictions, and thousands of *Media Objects*, the computation time and the use of resources (specially CPU but also memory handling the states) need to be reduced as much as possible. During the evaluation of the tool we also have found some performance problems when using big numbers in the model, due to the fact that the traditional μCRL representation of the natural numbers, based on the successor constructor, is not efficient when evaluating the model symbolically.

As solutions to this performance limitations, we are already exploring the use of a kind of theorem proving tools that are part of the μCRL `toolset`. As mentioned in Sect. 9.2.1, these tools are able to automatically prove certain properties of the μCRL specification, like confluence of a certain pair of actions. By exploring these properties the μCRL specification can be modified to a specification that results in a smaller, but observationally bisimular equivalent state space. The first experiments have shown promising results in reducing the size of the whole state space of the system (both in number of states and transitions). A new more efficient representation of the natural numbers is also being explored as a solution to the performance limitation in numeric computations of the model.

9.3.2 Final results: properties we are able to extract

9.3.2.1 Extracting global properties

For illustrative purposes, we have already included the small example state space generated for the `Configuration 1` in Fig. 9.5. The extraction of the properties takes the order of milliseconds and in all the cases it works without special problems:

- *Shortest fail* is obtained by model checking with the μ-`Calculus` property `[true*]<not 'fail.*'>true` and it gives the counterexample composed by the path `play(tape,m3,2)`, `play(cd,m2,2)` and `play(cd,m2,1)`. The path first explores the usage of the whole capacity of the tape device and then uses the capacity of the CD, asking twice for the same kind of movie, only available there. The counterexample can be seen as a graph with the `CADP` visualization tools (it is a subgraph of the original state space).

- *Longest general success* is extracted also using model checking with the following μ-`Calculus` property:
`[true*](<'fail.*'>true or <true>['fail.*']false)`. The model checking gives a counterexample of two levels, including `play(cd,m1,1)`, which leads to a state where failing is still impossible, and then a group of possible plays leading already to states with fails. So at most after two plays, the system can already give a negative answer to a user request.

- *Maximum number of plays* uses a logarithmic search and finds, in about a second, that in this case the maximum possible plays is 3. We can also analyze with the same kind of search the maximum number of plays for each of the movie types, giving a result of 3 times for $m1$, 2 times for $m2$ and 1 for $m3$.

- We have used for this configuration several examples of scenarios where we want to check if after playing a list of *Media Objects*, we still can play another list. We use the GUI we have developed for constructing the properties and the model checking takes the order of milliseconds in all the cases.

For the other two configurations, all the formulas have been used in a similar way obtaining interesting results. Checking if the extracted information is correct is less trivial because the reduced graphs for the second and third examples are already very difficult to print or see in a readable way. Still, for clarity reasons we are using a quite simple *VoDKA* configuration, and the results can fortunately be checked with what the human analysis of the system would expect. It is important to state that our tool is able to operate with more complex examples where the human analysis would be impossible or too difficult and costly.

For `Configuration 2` an example of shortest path to a fail state is described by the following counterexample:

```
<initial state>
"play(cd,m2,1)"
"play(tape,m7,2)"
"play(disk4con,m1,2)"
"play(disk4con,m6,1)"
"play(cd,m2,1)"
```

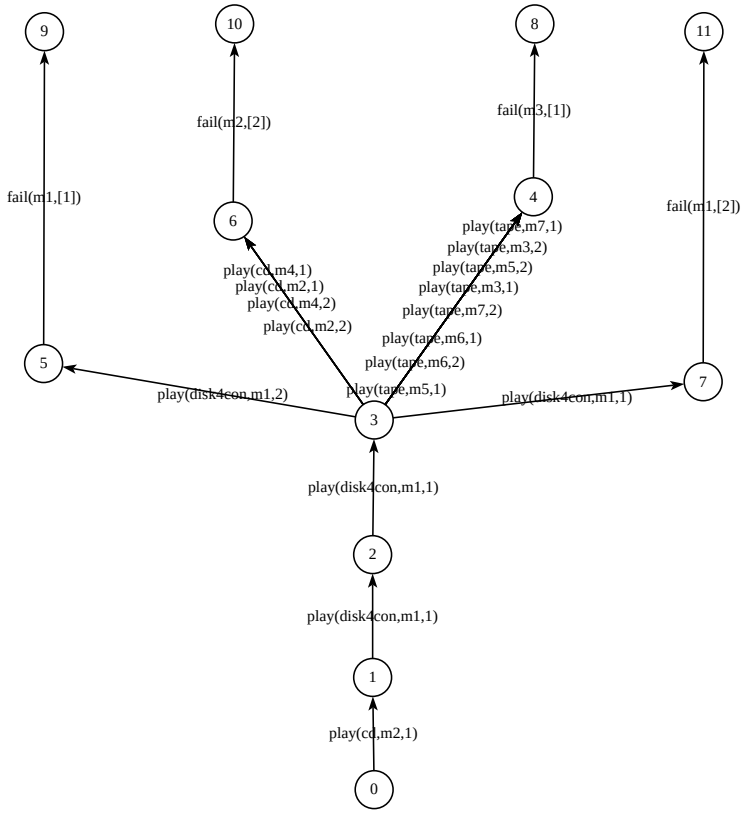


Figure 9.8: Counterexample for the longest success in *VoDKA* `Configuration 2`

```
"play(disk4con,m7,1)"
<goal state>
```

The longest general success for the same configuration is composed by a path with three plays, and the result is obtained as the counterexample in the graph that can be seen in Fig. 9.8. After those three plays, we only can do actions leading to a state where at least one fail is possible.

The maximum number of plays is 7, as one would expect from the analysis of the restrictions of the different storage devices, and the maximum number of plays that can be accepted for each kind of movie is also limited by the capacity of the devices storing that movies. The time it takes the logarithmic search for this size of the graph is in all the cases less than 3 seconds.

For `Configuration 3` the shortest path to a fail is described with the following counterexample:

```
<initial state>
"play(cd,m10,1)"
"play(disk5con,m1,1)"
"play(tape,m15,2)"
```

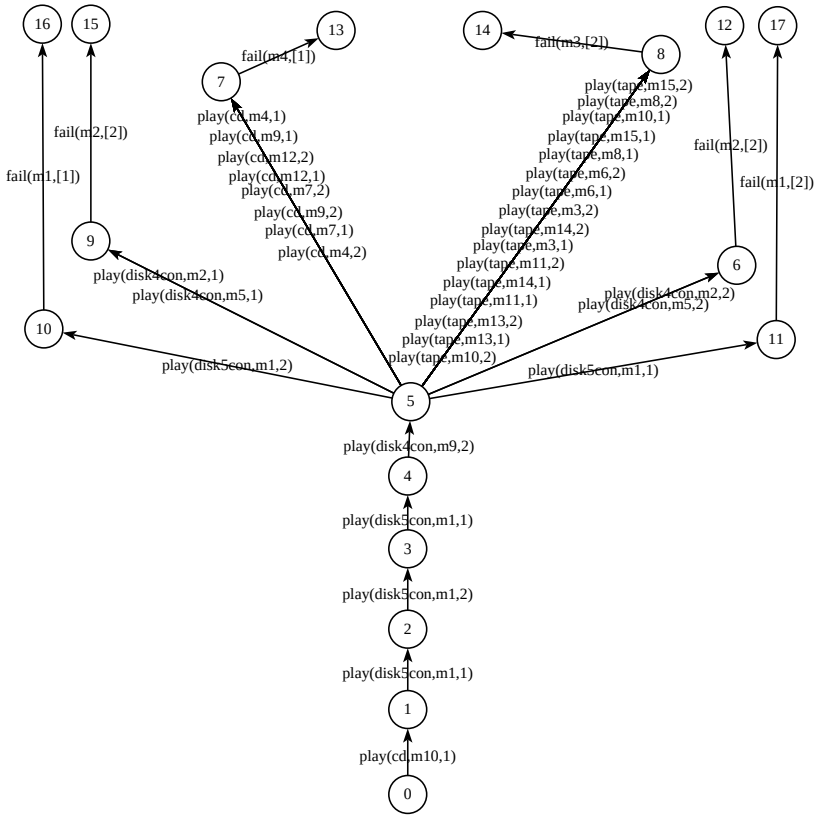


Figure 9.9: Counterexample for the longest success in *VoDKA* `Configuration 3`

```
"play(disk5con,m1,2)"
"play(cd,m7,2)"
"play(disk5con,m6,1)"
"play(disk4con,m15,1)"
"play(disk4con,m15,2)"
"play(disk5con,m1,1)"
"play(disk5con,m6,1)"
"play(disk4con,m15,1)"
<goal state>
```

The longest general success is five plays, and the result is obtained as the counterexample in the graph that can be seen in Fig. 9.9. After those five plays, again we only can do actions leading to a state where already a fail is possible.

The maximum number of plays is in this case 12, again as one would expect from the analysis of the restrictions of the different storage devices, and the maximum number of plays that can be accepted for each kind of movie is also limited by the capacity of the devices. The time it takes the logarithmic search is in all the cases less than 8 seconds.

9.3.2.2 Extracting architecture from the messages

The architecture graph was defined as a directed graph where the states are processes and the labels show their communication (they represent messages).

For each of the different configurations,the obtained graph is very similar, and can be created with the AUT file to AUT file transformation already explained in the method part of this chapter. The input to the tool is the complete state space, not the reduced one, because we are in this case interested in all the messages, including the internal ones.

For `Configuration 1` of *VoDKA*, the new AUT file obtained is shown in Fig. 9.10. The graph does not show all the real messages included in the original graph, but an abstraction of them. The tool accepts as input one function abstracting messages using pattern matching. Messages are grouped to a generic tag (in general it is possible to map them to any *Erlang* term). In our case, the abstraction has been performed using the following function:

```
filter({lookup,_,_}) ->
    lookup;
filter({lookupAns,_}) ->
    lookupAns;
filter({lookupAns,_,_,_}) ->
    lookupAns;
filter({play,_,_,_})->
    play;
filter({play,_,_})->
    play;
filter(Message) ->
    Message.
```

This capacity of the extension for drawing architecture graphs in the `etomcrl` tool has resulted very useful for making the obtained graph more readable. In this case, the number of labels is reduced from 2747 in the original complete graph to 23 in the abstracted architectural graph.

Both for `Configuration 2` (the abstraction goes from 69888 to 30 labels) and `Configuration 3` (from 2655048 to 37 labels) the obtained state space is very similar, with a much bigger reduction and only one or two processes more and a few extra messages than the smaller example.

For the biggest configuration, the translation to the architectural graph takes about 7 minutes. This can be a limitation for bigger state spaces, but it can be reduced by detecting when no more labels are generated after some number of messages. If a given number of continuous messages (the number being a parameter given by the user of the tool) is abstracted to labels that are already in the graph, and now new graph is generated, the generation of the architecture graph is stopped. It could be, if the parameter is not big enough, that some of the message patterns are not recognized, and therefore the graph is incomplete, but normally this is solved with experience and the amount of time needed for generating the graph can be reduced dramatically.

The fact of having a result graph in AUT format is interesting. We can use all the tools available in the `CADP` package for showing, changing or even executing

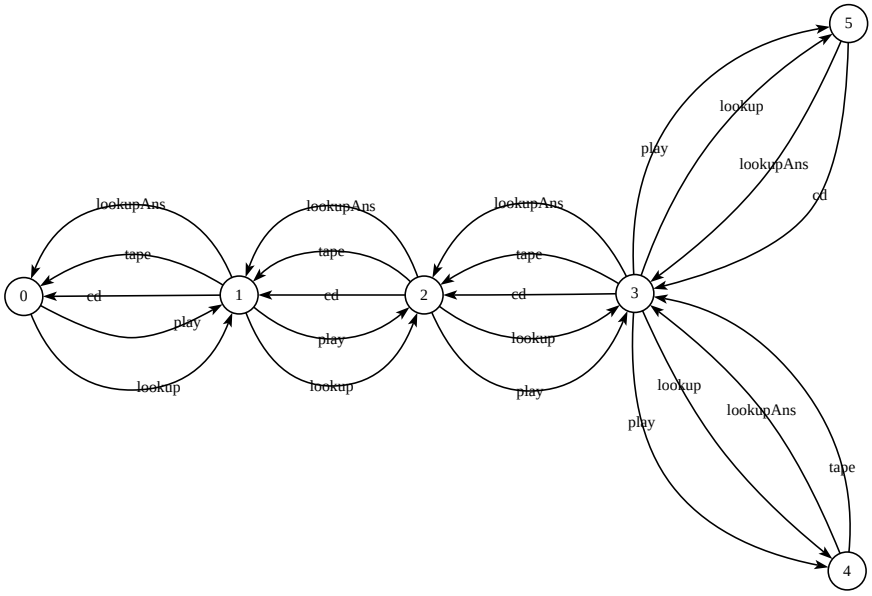


Figure 9.10: Architecture graph for *VoDKA* `Configuration 1`

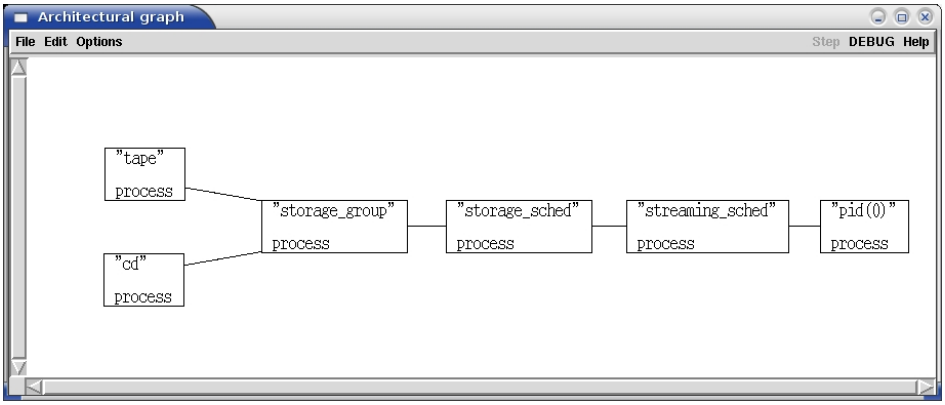


Figure 9.11: Architecture graph generated from the AUT file

(with the interactive GUI) the state space. We could even perform, using SVL, reductions similar to the ones shown above in the transformation tool.

From the AUT file (where the states are numbers) plus the dictionary (where the numbers are associated with process names or identifiers) we can use graph generation tools for creating a picture like Fig. 9.2.3.2. An example of what can be done is shown in Fig. 9.11.

9.3.2.3 Extracting bottleneck information

In the description of our method, we have introduced three kinds of system bottlenecks: internal independent bottleneck, user or external independent bottleneck, and relative bottleneck. We will discuss now how to extract the information about them from the system state space graphs.

For the **external bottlenecks** we can first see the reduced state space for `Configuration 1` shown in Fig. 9.6. Which one is *the point in the system architecture that makes the first fail to be answered to the user?* Using human reasoning, we can deduce it from the labels and our knowledge of the system architecture obtained in previous sections. The first fails are seen when we ask for $m3$ for the second time (with any bandwidth), and when we ask for $m2$ with bandwidth 2 after asking first for any of the movies of the CD device (i.e. $m1$ and $m2$). The first case is due to the connection limit in the tape device, and the second one to the bandwidth limit in the CD. But how could we extract this two bottlenecks automatically from the graph?

The solution we have selected is: in the complete graph, we find the shortest fail as we did for the reduced graph, but in this case we express the property trying to find a fail message sent to the user process (not all the fail messages are system fails now). As the counterexample, we get the path to that fail. Then, in the counterexample, we search for the first fail message of any kind, and we check which process is producing it. That process is the external bottleneck (meaning the bottleneck seeing from the user point of view).

If we run again that algorithm using the μ-`Calculus` formula:
`[true*]<not 'reply(pid(0),.*,.*,[].*'>true` in the `Configuration 1`, we get

the following counterexample (internal actions and assertions have been removed for clarity reasons):

```
<initial state>
"call(streaming_sched,{lookup,m3,[1]},pid(0))"
"call(storage_sched,{lookup,m3,[1]},streaming_sched)"
"call(storage_group,{lookup,m3,[1]},storage_sched)"
"call(tape,{lookup,m3,[1]},storage_group)"
"reply(storage_group,{lookupAns,[{1,1,tape}]},tape)"
"call(cd,{lookup,m3,[1]},storage_group)"
"reply(storage_group,{lookupAns,[]},cd)"
"reply(storage_sched,{lookupAns,[{1,1,tape}]},storage_group)"
"reply(streaming_sched,{lookupAns,[{1,1,tape}]},storage_sched)"
"reply(pid(0),{lookupAns,m3,[1],[{1,1,tape}]},streaming_sched)"
"call(streaming_sched,{play,m3,1,tape},pid(0))"
"call(storage_sched,{play,m3,1,tape},streaming_sched)"
"call(storage_group,{play,m3,1,tape},storage_sched)"
"call(tape,{play,m3,1},storage_group)"
"reply(storage_group,tape,tape)"
"reply(storage_sched,tape,storage_group)"
"reply(streaming_sched,tape,storage_sched)"
"reply(pid(0),tape,streaming_sched)"
"call(streaming_sched,{lookup,m3,[1]},pid(0))"
"call(storage_sched,{lookup,m3,[1]},streaming_sched)"
"call(storage_group,{lookup,m3,[1]},storage_sched)"
"call(tape,{lookup,m3,[1]},storage_group)"
"reply(storage_group,{lookupAns,[]},tape)"
"call(cd,{lookup,m3,[1]},storage_group)"
"reply(storage_group,{lookupAns,[]},cd)"
"reply(storage_sched,{lookupAns,[]},storage_group)"
"reply(streaming_sched,{lookupAns,[]},storage_sched)"
<goal state>
```

Now we analyze the first internal fail messages (i.e. `lookupAns` messages with the empty list as streaming options), we would see a very early fail in the CD device due to the absence of that *Media Object* there. And then we detect two more fails, one again with the CD and the other with the tape. Including knowledge about the presence of the movies in each device in the automatic algorithm, we can detect that the resource limit in tape together with the absence of the *Media Object* in the CD. With that information, the system designers would be able to decide if they want to add new resources or change the allocation of *Media Objects* in the devices.

In order to make easier the bottleneck analysis, it would help a lot to have different messages when a movie is not in the system from the ones that are sent back in the protocol when the *Media Object* is there but the resources are not enough. Thinking from a more general point of view, it would be interesting also for improving in the future the internal scheduling of the system (decisions could be taken at a higher level for example for moving a *Media Object* if it is not present). So this is left as a suggestion for the system designers that shows also the kind of feedback that can be obtained from this kind of system analysis.

For the other two configurations of *VoDKA* we are studying, the results were comparable. In the `Configuration 2` the bottleneck that is found is the tape

when the users request $m3$, only present in the tape, twice. Similarly, in the `Configuration 3` it is also the tape device when the users request first $m11$ (present in several devices but the tape is selected), and then $m3$, only present in the tape device. As the maximum number of connections is set to one, the fail is produced as the first one in the system. Two feedback messages are obtained: the tape is an important bottleneck; and this could be less important if the cost of the tape is increased so that it is the last device that would be selected if there are other alternatives.

However, the time it takes to generate the counterexample with model checking goes up to several seconds for the more complex example we are considering, which means that it could be a potential problem for more complex configurations. A solution for this is to avoid generating the whole state space, trying to create only the part of it that is relevant for our interests. In order to do that, we can use the extension to the `etomcrl` tool for observing special messages stopping the graph generation when that messages appear.

For our example, we just need to introduce the following header in the modules we want to study (in our case we selected the storage group as the key point for our analysis):

```
-observe_messages({bottleneck_message, ["{lookupAns,[]}"]}).
```

The annotation in the *Erlang* code should be read as: if this module produces an answer with empty options (meaning the *Media Object* cannot be played from that component), the state space generation should be stopped and no further paths from the current state should be generated. Besides, the label going in this path to the last node, should contain the message identifier `bottleneck_message`.

We can express now easier properties (using this special label just introduced in order to detect the first occurrence of it: the shortest path to a fail) and what is much important from a practical point of view, the generated graph in our case the reduction is not very big, but depending on the frequency of the fails in the original graph it can increase a lot.

For the **internal independent bottleneck** the process is simpler than above. We just need to find the first fails in the system, no matter if they cause an error for the user or not. This can be done also by counterexample extraction, but now we do not need to specify that we want to find fails that are actually sent to the users. For the three configurations, we have done it and we obtain what would be expected: the devices with less resources are the ones showing the first system fails, together of course with the fails produced by the absence of the requested *Media Objects*. The issues with the performance for bigger graphs are again solved by using the extension developed for observing messages and stopping the graph generation with `etomcrl`. This time, as we can observe messages directly in all the components, the reduction is much bigger, and we can easily go from a graph with several million states to another one with at most several thousand.

Finally, the **internal relative bottleneck** are related to distance between fails inside the system. In the case of *VoDKA*, this kind of information is specially complex to obtain. We can easily specify a property saying that after a fail due to lack of resources in the CD device, there cannot be N plays streamed from the tape device. If it is the case that $m1$ is only in CD and $m2$ is only in the tape, then

after streaming $m1$ as many times as needed for provoking a fail in the CD, we still are going to be able to stream $m2$ from the tape up to the tape capacity (assuming there are no other bottlenecks around). Should we interpret from this scenario that CD and tape are unbalanced and that the tape is a relative bottleneck? It is clear that the answer is no.

The solution is to reformulate a bit the way of finding this kind of bottlenecks in our system: we should only look into what happens when we ask for the same *Media Object*, and only in the group of devices that contain that *Media Object*. In the previous example of tape and CD, if both contain $m3$, and after answering to the user a fail for $m3$ from the CD, we still can play it N times from the tape, that could (it is only a possibility) show that the system scheduling is not well balanced and something should be reconfigured or reallocated. We have applied this approach to our three configurations and the results were satisfactory, obtaining the expected answers: as the algorithm we were considering selects always the device with less cost, ant cost is not changing depending on the load of the device, so the system is unbalanced. Changing the *Erlang* model so that the devices are selected randomly or so that the cost is increasing depending on the load, makes the checking return a more balanced status (i.e. with less presence of paths suspicious of being bottlenecks).

9.3.2.4 Adding and studying new components

For easily illustrating the approach we propose for obtaining information, lets consider that we want to add a new disk device to the `Configuration 1`, without changing any other thing. Given that the rest of the restrictions are not going to change, we want to know how many connections and how much bandwidth we should add to the new component in order to maximize the system capacity, and in order to avoid to convert the new resource in the system bottleneck.

A very clean solution is adding the new component with infinite resources, representing that by a reasonably big number extracted after analyzing the architecture (doing this automatically is left as future research). In our example, we can add it with maximum number of connections 50 and maximum bandwidth 100, which is clearly *enough*. We then generate the new state space, which in any case is not going to be much bigger than before, because the resources in the rest of the components are limited anyway. The new bottleneck of the system is going to be new in the upper levels of the hierarchy: in concrete the 10 connections and 15 units of bandwidth of the `storage_group`.

Now we need to extract by graph analysis the information about the maximum number of times that the new component is going to stream a movie and the maximum number of total bandwidth that it would need in the "worst case".

In our example, the worst cases would be two: the first one, when ten movies are played from the new device, and the second one, the path were a number of movies using up to a total of 15 units of bandwidth are used. Those are the two limits imposed by the restrictions in the upper levels.

Again, how can we extract that information automatically from the graph? We propose to use the same approach we presented above in the logarithmic search, but now working with the whole state space and adapting a bit the logical expressions. We perform two searches: first, the maximum number of play messages from the

new device that can contain a graph; second, the maximum amount of bandwidth that can be contained in any path of the graph. The second set of formulas is a bit more complex to generate, because they have to take into account the bandwidths available in the *Media Objects* of the system.

Assuming that we have the labels renamed to play and fail, and that fail has three arguments (*Media Object*, bandwidth and the device from which we are going to play), the first property looks like, where N would be the variable that changes while searching:

```
<true*.'play(.*,.*,device)'.true*.'play(.*,.*,device)'...
        ---- we repeat N times the expression ----
        ...'play(.*,.*,device)'>true
```

For the second property, we would use a similar formula varying both N and the bandwidth argument, which would take all the values of the bandwidths present in the system.

With the feedback obtained, the system designers would know that adding to the new device more resources than 10 connections and 15 units of bandwidth would not increase the global capacity of the system.

9.4 *VoDKAV*: hiding formal methods in the analysis

As one of the goals when designing the framework was making it available to the developers, it was very important to be able to hide all the theoretical and technical tools and techniques used from the end user. The role of the GUI developed in *Erlang* and called *VoDKAV* (standing for the VoDKA *Verification* tool) is to make it easier for the user to handle different models and configurations, and to go for each of them through all the steps of the method we propose.

The tool can be seen as a proof of concept for which a prototype was developed: is the method simple enough for being useful directly to the developers? If we are able to develop a tool that hides as much of the formal concepts as possible, the answer could be closer to yes.

In this section we talk about the goals of the tool and in Appendix C the most important details about its design and implementation can be found.

The first goal of the tool was to make easier the management of the *VoDKA* models and the system configurations. The *VoDKA* source code is not specially stable, it changes in time and gets more complicated, and we want to handle all this versions at the same time. Fig. 9.12 shows how this is solved. The user can add, modify or delete designs, each of then containing, apart from the meta-information, a pointer to the folder where the *Erlang* model is implemented, and another pointer to the folder where the system configuration is described.

The second goal of the tool was to allow the user to change the configuration for a given design and then easily go through steps 1 and 2 of our method, i.e., the generation of the μCRL specification and the creation of the complete and the reduced state graphs. Fig. 9.13 shows the windows where this can be managed: the user can change the system architecture (adding a new component or changing the way the components are connected), change the resources available (bandwidth and connections) and the cost for each component, or changing the *Media Objects*

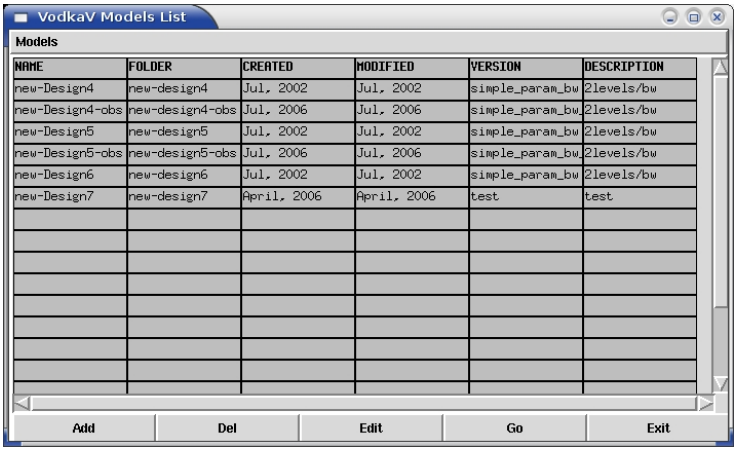


Figure 9.12: *VoDKAV* graphical user interface: designs repository

present in each device with just a few clicks. In order to check the configuration, some complementary functionality like the drawing of the supervision tree is provided, as can be seen in Fig. 9.14.

Once the configuration is ready, with just another click all the information is extracted and prepared for giving it as input to each of the tools we are using (`etomcrl`, `McErlang`, μ`CRL toolset`, `CADP`, etc.). The user can generate an architectural graph, or create and view the reduced state space, for example. No knowledge on the underlying tools is required, and everything is done automatically.

Third and last goal of the tool is to make easier the extraction of information from the state space, i.e., the third step of our method. This is solved, as can be seen in Fig. 9.15, with a window that hides the creation of the μ`-Calculus` properties as much as possible, where the user can execute the properties and see the counterexamples. If the user is advanced and want to define its own property, it still can be done, but the most frequent ones are encapsulated and described in a language very simple to understand.

The tool has been extensively used during the latest years for the development of the thesis, saving a lot of time and avoiding to deal all the time with the details related to the formats, interfaces and languages of all the tools we are using in the method.

9.5 Testing the method using `McErlang` as model checker

In section 8.5 we explained the `McErlang` tool, which generates a state graph and model checks a property on-the-fly directly from the *Erlang* source code. In this section we will explain how this tool fits in the method we have presented for going from the software architecture to the formal verification of a distributed system. First we explain our experiments for generating the state space directly from the source code instead of first translating to μ`CRL`. After that, we show how the capacity properties can be checked using `McErlang`. Finally, we compare the

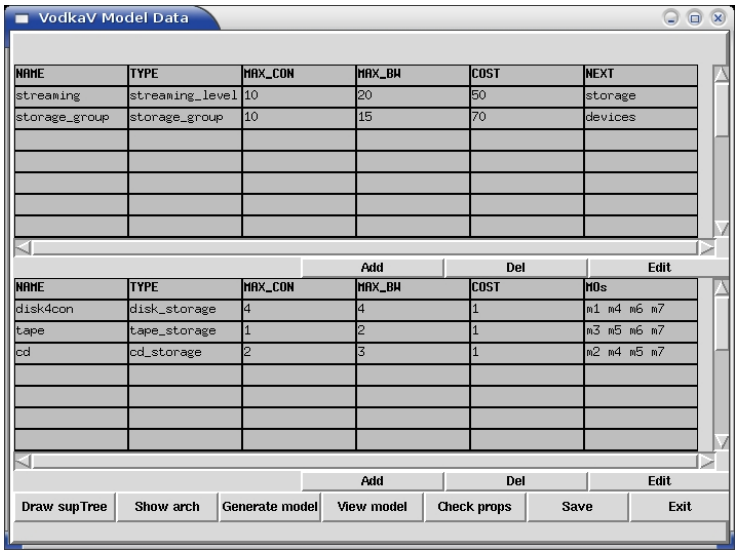


Figure 9.13: *VoDKAV* graphical user interface: model manipulation

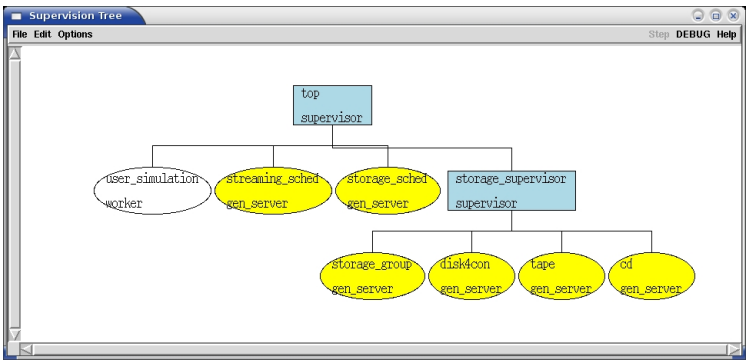


Figure 9.14: *VoDKAV* graphical user interface: supervision tree

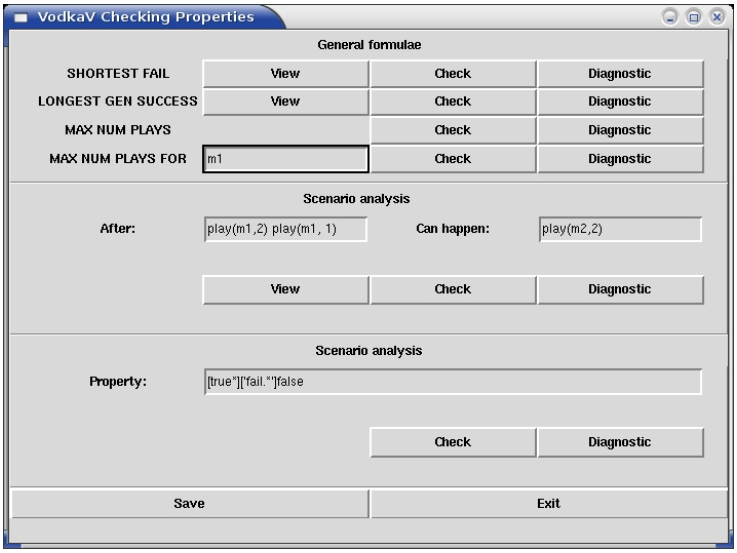


Figure 9.15: *VoDKAV* graphical user interface: property checking

results obtained with those already presented for the other tools.

9.5.1 Generating the state space from the *Erlang* model

In order to use McErlang for generating the complete state space for *VoDKA*, we needed to provide the *VoDKA* source code, an abstraction, a hash table implementation and a monitor implementing a trivial one-state correctness property (in order to avoid influencing the size of the resulting program state space).

For the graph generation, we used the default abstraction (it only orders the processes using their identifiers), and also the default hash table, which stores the states in a *Erlang* ETS table. As monitor we implemented a trivial one-state function always returning ok with the monitor state unchanged.

For the source code we used exactly the same version that was translated to μCRL, but now needing to translate it by hand to the internal notation of McErlang. This was quite straight forward and can be repeated following the steps already explained in section 8.5 (in the future it will be automated as part of the improvements that are planned for the new versions of McErlang).

The only difference was the modelling of the users. As we explained in our method, we use only one μCRL process in order to simulate a user asking non-deterministically for all the possible combinations of *Media Objects* and bandwidth. This was done in the translation to μCRL because *Erlang* lacks a non-deterministic construct.

If we consider the following client code:

```
loop(StreamingScheduler) ->
   {lookupAns, MO, Profile, Options} =
      gen_server:call(StreamingScheduler,{lookup,m1,[1,2]}),
   case choose(Options) of
       fail ->
```

```
        loop(StreamingScheduler);
      {Pid, Bw} ->
        ...
  end.
```

With McErlang, the non-deterministic choice return value can be used in order to directly model the *user process* in (pseudo) *Erlang* in the following way (for a system with 15 kinds of movies and two kinds of bandwidth):

```
loop(StreamingScheduler) ->
    {choice,
     [{?MODULE,select_movie,[{lookup,X,[Y]},StreamingScheduler]}||
         X <- [m1,m2,m3,m4,m5,m6,m7,m8,m9,
               m10,m11,m12,m13,m14,m15],Y <- [1,2]]
     }.

select_movie(Message,StreamingScheduler) ->
    ev_gen_server:call(StreamingScheduler,Message,
                       {?MODULE,loop1,[StreamingScheduler]}).

loop1([StreamingScheduler],{lookupAns, MO, Profile, Options}) ->
   case choose(Options) of
       fail ->
         loop(StreamingScheduler);
       {Pid, Bw} ->
         ...
   end.
```

The size of the state space generated lies in the middle of the complete state space generated from μCRL, and the reduced one presented in the previous sections after hiding the internal messages of *VoDKA*. It contains less internal actions and assertions than the whole state space, so it is smaller, but the transitions represent all the messages that are received all over the software architecture.

When we started to use McErlang for *VoDKA*, some irrelevant messages were increasing the size of the graph. This is the case of the exit messages that some of the intermediate processes used in order to start the supervision tree where generating when finishing their execution. These messages could be received, by their supervisors, in several places in the graph, and this was increasing the final graph without giving any interesting information for the properties we were going to analyse. We managed to abstract from this by slightly modifying the source code of the *Erlang* examples in order to keep these processes from finishing (the changed processes wait forever on the reception of an impossible message, and thus do not influence the state space generation).

- Configuration 1:
  - Size of the state space: 1283 states explored, accessed states 1207, hashed states 1207 (half the size of the state space obtained with μCRL).
  - The generation time for the state space is less than one second (with μCRL was almost 8 times more).

- Configuration 2
  - Size of the state space: 30751 states explored, accessed states 28995, hashed states 28995 (about 2 times smaller than the obtained with μCRL).
  - The generation time for the state space is less than 14 seconds, about four times less than the time spent with μCRL.
- Configuration 3:
  - Due to the very big size of the state space, the generation of a graph with several million state spaces requires too much memory at the moment. Optimising the memory consumption of the McErlang tool is work in progress. Instead, we use this example to demonstrate how the current limitations can be worked around thanks to the flexibility of the tool. We show later in this section how we extracted some properties from this graph without needing to generate it completely.

We have shown earlier that for some properties it would be interesting to use a reduced version of the graph, where the internal messages of *VoDKA* would be hidden, and with only the *play* and *fail* messages present. This graph is smaller, easier to handle, and still useful for checking some of the capacity properties as seen in previous sections.

In order to go from the detailed state space to the reduced one, using McErlang, in this case we need to write an specialised abstraction module. Concretely the solution is to write a *normalise* function making all the states where a message is going through the system to appear like the same state, i.e., if a lookup message is going from the users all the way down the system architecture till the storage devices, all the states where that message is moving through the system should be mapped by the abstraction to the same one. This can be done by creating a normalised state where the messages are all removed from the system queues and placed in a set of messages pending to be received, including all the messages for all the processes.

Once we have the complete or reduced graphs generated with McErlang, we can easily generate them in a format understood by CADP, and then model check our properties as explained in the previous sections. We would already gain something, as the state space would be already smaller, and its generation more optimal CPU-wise. We could also generate the architectural graph as we shown in previous sections. However, McErlang is on itself a model checker and checking the properties at the same time new states are generated adds more advantages to the approach, as it will be explained in next section.

We have also explored the feature of McErlang to take into account the channel delay semantics. However, with smaller configurations the generated state graph was already composed by several million states. For the kind of properties we are exploring here, the results were obtained without taking into account channel delays.

9.5.2 Checking the properties from the *Erlang* model

In order to illustrate the differences with the approach of using `CADP`, we have selected one of the capacity properties shown in the previous sections: the shortest path to a fail in the graph.

To extract that information, we needed to write a special monitor for `McErlang`. The internal state of the monitor was not used, but still the monitor was made aware of the minimum counterexample already found in the part of the graph we have explored.

For each of the states, the monitor function is called from the model checker. Then, the monitor explores the list of processes in the system and identifies the ones corresponding to users that are about to receive a message from the system saying that the requested *Media Object* cannot be streamed. That process state identifies a fail in the graph, and the monitor can then check the size of the counterexample to determine if it is the shortest counterexample seen so far which should thus be recorded[2], or otherwise, whether it should be ignored. We have extended `McErlang` with a simple function that, when the model checking finishes, prints the counterexample that was stored as being the minimum path to a fail.

Running the three configurations with the monitor described gives as answer the shortest counterexample to a fail after generating the whole state space. The results are obtained a bit faster than with the μ`CRL` approach for the configurations 1 and 2, but we still have limitations with the third configuration due to the size of the state space.

In Section 9.3.2.3 we explored the idea of using the *observe_messages* `etomcrl` extension in order to reduce the size of very big state spaces only generating the part of them relevant for our properties. The basic idea was to generate a deadlock when a given message was received. With that approach, we managed to reduce the state space, but the reduction had some limitations. Using `McErlang`, due to the flexibility of the tool which means that we can access all the information in the global state about the state of each of the processes running in the system, we improved the graph reduction dramatically.

The solution selected was to implement a variation of the standard hash table. In our modified hash table, we modified the implementation of the function `permit_state` which is called by the model checker to determine whether a newly generated state has been previously explored. The modification work by cutting search (i.e., declare that a new state has been previously seen) if the path from the initial program state to the new one is longer than the smallest counterexample seen so far. This means that, for example, if the smaller counterexample we have already found is 50, we will not explore anymore any states reachable only on paths of length greater than 50.

The percentage of the graph that is now explored depends directly on when in state space exploration carried out by the depth first algorithm used by `McErlang` the counterexamples appear. In the case of the first configuration of *VoDKA*, we reduced from 1207 to 935 states. For the second configuration, the reduction is already bigger, and goes from 28995 down to 5355 when using the new hash table (and obtaining the property takes now less than 4 seconds instead of the previous

[2]The shortest counterexample is kept in a global variable during model checking.

14). In the case of the third configuration, the reduction is even bigger, and we manage to extract the capacity property of the shortest fail in a matter of 80 seconds, with a graph of size 42239, very small compared with the original one that had approximately several million states.

The example above shows the flexibility and power of the McErlang tool. The same approach works fine for other kind of properties we have verified using CADP and μ-Calculus. The longest success, the maximum number of plays and other similar capacity properties can be verified in the same way: developing a monitor that stores the size of the maximum path already found. Unfortunately, when the property searches for the maximum path we cannot reduce the size of the graph with a special hash table. Scenario-based properties are also easily specified by simple *Erlang* monitors.

The flexibility of the McErlang tool can be also used in order to extract bottleneck information or studying the resources needed to add a new component. The procedure would be the one already exposed in the method section, but due to the access to a more complex state and action information we can extract and take into account even more variables.

Finally, we have explored the use of McErlang for other kind of properties not directly related to capacity. An example of this is checking if the user always receives an answer after sending a lookup message.

In order to verify that, we implemented a new monitor where there are two internal states, one *dangerous* state where the user already sent the message and there is an answer message pending to be received, and the state where no answer is pending. The property can be tested by introducing a choice in one of the messages of the *VoDKA* system where one of the options is not sending a message back, and it works without problems for our system. Unfortunately liveness properties such as the response example here cannot currently be checked by the tool in the presence of cycles in the underlying state graph (not present in the *VoDKA* example), although standard algorithms to do so exists.

9.5.3 McErlang vs. etomcrl +μCRL +CADP for *VoDKA*

We have shown that McErlang can be used both for generating a state space similar to the one obtained with the μCRL toolset, and for model checking the properties about the *VoDKA* software architecture in a similar way to CADP.

McErlang lacks some of the features found in mature tools like μCRL toolset and CADP, but it has some advantages due to the approach of working directly on the *Erlang* level. Apart from the general features already described in Section 8.5, we have found some advantages for the case of using McErlang with *VoDKA* for extracting capacity information.

The main advantage comes from the fact of being able to analyse the real state space of the system in detail, accessing *Erlang* structures, message queues, and so on. This is more powerful than reasoning only about actions and states without information as it is the case with μCRL toolset and CADP.

The flexibility and extensibility of the McErlang tool also allows the user to implement sophisticated abstractions and hash tables. With μCRL toolset we needed to generate the whole state space before reducing it later with CADP. Changing the *Erlang* model including *observe messages* examples, we managed to reduce a bit

the size of the original one. But now, with McErlang, we can stop the graph generation in a more flexible way whenever we are in a path where we already know that the next states are not relevant for the property we are trying to check. Doing this, we have been able to extract properties from state spaces that where too big for being generated completely.

9.6 Analysis and discussion

In this chapter, the scheduler of the *VoDKA* system acts as a case-study for our methodology to verify global properties of a system. The behaviour of the system is hidden in a complex distributed scheduler, based on component restrictions (bandwidth, number of connections), local policies (cost functions, filters), cost related functions (state of the components and resources still available), and a flexible hierarchical architecture. By means of this scheduler, we present a methodology for extracting global properties of an *Erlang* system from the local restrictions hidden in its processes. The methodology is more generally applicable than only for schedulers. Many systems can be seen as a deterministic function over the input (in our case sequences of users demanding movies). However, these functions are composed of many small functions: the components. These components can have state and are therefore hard to statically analyze. However, with our approach, we simulate all possible runs of the system, as such obtaining the function between domain and range compacted as a graph.

The methodology is based on three main steps that are all performed in a completely automatic way. In the first step, we use our Erlang *to μCRL compiler* to translate the original *Erlang* source code modules into a μCRL specification. In the second step, we use the *μCRL tool set* for generating the whole state space, which then is reduced to a smaller one where only the information needed for extracting the performance properties is shown. Finally, we use the model checker of the CADP tool set in three complementary ways for extracting interesting properties for the user from the reduced state space. All these steps are performed by the user with a high level graphical user interface, developed with the goal of hiding the internal details of the approach. The tool is even able to analyze the system in order to provide the user some feedback information with suggestions about how to improve the system performance.

At the moment our approach is the only one that can automatically verify this kind of global properties of a system. Other code level model checkers for *Erlang* with this accuracy do not exist. For Java and C, however, there are similar approaches when it comes to model checking source code [CDH00, HP00, Hol91]. We are not aware of an attempt to use these source code model checkers for analyzing a flexible concurrent and distributed architecture in the way we explain in this report. We apply ideas of simulation in a model checking framework.

There are many papers on the analysis of scheduling algorithms and simulation. We can state that most simulation tools use a simulation specification language that differs from the programming language the system is implemented in. We are not aware of an automatic translation from the source code to the simulation language.

The *Erlang* source code of the *VoDKA* scheduler contains approximately 800 lines of code distributed over nine modules (plus the code of the generic behaviours,

of course). This code implements the kernel part of the video server. We modified the code by abstracting away the parts that are not necessary for the performance analysis (e.g. low level transmission protocols and processes). We also ignore the release of resources, since we want to look at the overloading of the system. This means that we are looking at the worst case performance scenario where users only request and do not release a *Media Object*.

We want to obtain this state space directly from the source code of the system. In more classical approaches in literature one often uses a formal specification of a system, normally a manually constructed abstraction of the system. The advantage of the use of *Erlang* is that we have a rather high-level of abstraction already, such that we can use the source code as our starting point. However, even the *Erlang* source code would contain too many details to make it feasible to generate the full state space. We really build upon the existence of design patterns, like the supervision tree and generic servers that hide a lot of the details.

With the approach described in this report, we are able to handle configurations of the system that are as complex as the ones that are being used in the *VoDKA* prototypes that have been deployed for the cable company (with the manual abstraction that there single cache level is seen as a storage level). This means that we are able to extract automatically performance information about the system from its source code and configuration parameters.

The proposed approach uses formal methods in a rather original way: using the μ-`Calculus` as a powerful declarative graph information extraction language; and using model checking tools as general, flexible and efficient graph search algorithms. We verified properties for several configurations. Most time is spent on the generation of the full state space. The compilation from *Erlang* to μ`CRL` takes only a few seconds and similarly the reduction of the state space and verifying properties of it only takes a few seconds up to a minute. Generating the state space, though, may take a few hours for rather large configurations. For that reason, we are exploring the best way of using μ`CRL toolset` for converting the process algebra model in the system to a bisimular equivalent one that produces a smaller state space of the system. We can indicate already in the specification which action we want to hide and perform transformations on the μ`CRL` level to obtain a specification that results in a reduced state space. The first results with these tools are very promising, being able to reduce more than ten percent of the size of the state space. Additionally, a new representation of natural numbers in the process algebra, in order to improve the performance of the symbolic computation of the model, is subject of study.

We have also shown a complementary approach based on `McErlang`, a model checker that works directly with the *Erlang* source code, that already solves some of the problems seen in the initial alternative.

From the developers point of view, most of the formal methods tools and technologies are just research initiatives that are still far away from being useful for software development. In our case, since the beginning the focus was to try to avoid this, combining research with creating methods and tools useful for the project developers. The GUI developed for hiding the internal details of the approach has permitted to keep the research closer to the development team. Still a lot of work needs to be done, but the obtained results in this sense are very satisfactory.

9.6.1 Conclusions and future research paths

This chapter has shown a method for going from software architecture to the formal verification of a distributed system. The method is very structured and consists on several steps that are based on a group of mature tools we have selected or developed. We have illustrated the method by a case study: the extraction of performance information from the software architecture of the *VoDKA* system, a *VoD* server developed using *Erlang*. We have been able to automatically extract relevant information about architecture bottlenecks and system capacity, providing a tool that can be used by the developers in order to learn more about the system.

The method has some limitations, some of them due to the use of model checking, others due to the use of the selected tools, but it is very automatic and permits to introduce formal methods in the software development process in order to increase the quality of the system, as we had stated in Chapter 7.

As we have explained, the method can be applied to other distributed systems, with other architectures or even written in other languages. The idea of going from the source code and the system configuration to the model checking of capacity properties complements the classical approaches of using model checking for finding functional errors in the system.

Although the results have shown to be useful for the development team, we have done only the first steps towards a really mature inclusion of formal verification in the analysis of software architectures.

Some of the relevant lines that are being considered for the near future research are:

- *VoDKA* is having more and more users and new versions are developed. It would be interesting to learn more about which are the real differences between the *Erlang* model of the system we are using and the real prototypes that are being used currently. Concretely, would be interesting to find out how far we really are from taking the real source code as input of our methodology, even abstracting automatically from the low level details (automate the work we are now doing by hand).

- Consider the complete addition of the 'cache' level to the '*Erlang* model'. This is going to produce some big changes in the way we extract the properties from the system. This is going to have a lot of influence also in the way we are planning to use the abstract movies (movies that represent the set of *Media Objects* in all the devices, in only one, in device A and B, etc.)

- Explore new ways of giving the user feedback information. For example, implementing some algorithms that use the abstract *Media Objects* for giving this kind of feedback. What happens if we want to give feedback information that needs different state spaces of different models? How can we really use this different state spaces for advising the user on how to change the system restrictions? It is in particular interesting to be able to extract information about bottlenecks of the system, i.e. being able to give the user more complete answers (e.g., "Thirty users cannot request Star Wars simultaneously with the proposed configuration of the system, because the bandwidth communicating the CD device with the system is too narrow").

- Understand better which properties the users of our methodology would be interested in. A lot of discussion has happened already, but we could do a more systematic study with the current *VoDKA* Team and possibly with some users of the system, a complete list of user-readable properties that then we could try to extract from the local properties of the system.
- We are considering also to explore the use of 'compositional model checking' as a way of dealing with state space explosion also in our methodology for performance analysis.
- Continue with the experiments related to using the theorem prover like tools of the μ`CRL` toolset that can be used for reducing the size and time in the generated state space. Important for being able to handle bigger examples.
- Study which features would the users need in the GUI and how to improve it in order to hide even more details of the underlying method and tools.

But our future research goals are more ambitious. An interesting feature would be to reduce the need of manually abstracting from some parts of the *Erlang* source code. Ideally, the real system should be directly used for the model checking. Using `etomcrl` it is difficult to do that, because the approach needs inherently some kind of abstraction, but `McErlang` is much closer to the *Erlang* program and has a more flexible nature, so we will increase the efforts for getting new results in that direction.

In general, we want to built upon this research to create a set of tools that help future software architects to evaluate the quality of their architectures in a early phase and iterative in the development process. This means that we need to continue improving our understanding of what software architects need to get from our tools, and at the same time increase the maturity and degree of automation of our methods and procedures.

Part IV

Conclusions and open paths for future research

Chapter 10

Thesis conclusions

In this thesis, we have shown how to go from the software architecture to the formal verification of a distributed system. The target of our research was *VoDKA*, a very flexible, scalable, distributed *VoD* server developed by the *LFCIA-MADS* research group during the latest years using *Erlang/OTP* technologies. We have developed a method and several tools for applying formal methods to the target system. The goal was to extract useful architectural information automatically from the source code and the system configuration.

The thesis was divided into three main parts. Although the results and conclusions of each of the parts were detailed and discussed already during each of their chapters, we will try to summarize the most important ones hereafter.

In the first part of the thesis, we gave a general motivation for the research: a very innovative distributed system, sharing some interesting features with all the systems of its kind, was a very challenging target for studying how formal methods could help to improve the software architecture. Also, the goals of the thesis were described, mainly: to obtain a better understanding of the software, to propose tools for improving its architecture, and to study the possibilities and limits of formal methods for verifying non functional aspects of a distributed system extracting automatically properties about it. Finally, in this part, the main concepts from the different areas that are related to the thesis are presented. We discussed specially the ideas referred to the development of distributed systems using *Erlang/OTP* and design patterns, and also to the use of process algebras and model checking for extracting properties from that kind of systems. This part did not have important conclusions and acted more as an introduction for the rest of the manuscript.

In the second part of the thesis, the *VoDKA* system was studied and described in detail. The goal of that part was to describe the software architecture and at the same time to show how several features of the system motivate the use of formal methods in order to have a better understanding of its architecture. We have discussed in detail which requirements are the most important ones when developing *VoD* servers, and also how the state of the art in the field influenced *VoDKA*; at the same time, we have explained why the innovative proposal made by the *LFCIA-MADS* group is a better option than the existent alternatives for scenarios where, among other features, flexibility, scalability and affordable cost are required. Using the *4+1 model*, we described the system from several complementary points of view. Guided by the *VoD* requirements and the use-cases, the

design of the system heavily uses design patterns for creating its main software components, which then collaborate in a very flexible way for building the software architecture. This could be easily done mainly because all the components share a common message API. The configuration of *VoDKA* can vary a lot depending on the concrete needs of each deployment. We have shown several different deployments and how *VoDKA* is configured in each of them. We have also described how the architecture of the *VoDKA* system has evolved at different levels during its life cycle. This, together with the lessons learned from the *VoDKA* development that are explained in Chapter 6, motivates the need for good tools for extracting information about the *VoDKA* system. *Erlang* lacks a type system, and although the platform is very powerful for developing complex distributed control systems, it also lacks good tools for helping the designers to make good architectural choices.

In the third part of the thesis, from the deep understanding of the *VoDKA* architecture, we explored how formal methods could be an answer to the lack of good tools for extracting information about distributed systems. Chapter 7 is itself an important result of this thesis: it explains and discusses why and how we decided to use formal methods for the *VoDKA* project. The main motivation was increasing the quality of the system. We decided to collaborate in increasing the quality by giving the *VoDKA* team a method for learning more about the software performance. Our approach included detecting design problems or finding errors, but was more focused on the automatic extraction of software architecture information from each version of the implementation. Inside formal methods, we selected model checking. Simulation, testing and theorem proving could be used as complements to our approach, having some advantages and disadvantages that we have detailed throughout the thesis manuscript. Our proposal follows an *agile formal verification* approach in each iteration of the software development life-cycle in order to enrich the design decisions for the future features that are going to be added to the system.

Our agile model checking-based approach, detailed and discussed in Chapter 9 consists in defining a method that, given the source code of our system and the architecture configuration as input, is able to extract performance information using logical formulas that are model checked against the system state space.

But for each of the steps in the method, we needed concrete tools. Chapter 8 discusses in detail the ones we have selected and developed after seeing the needs of our method. In our initial approach, we first go from the *Erlang* source code to the process algebra μ`CRL`; in order to do this, an *ad hoc* compiler was developed. The compiler, called `etomcrl`, uses the *Erlang behaviours* as a basis for simplifying the translation, abstracting from some low level details that are not relevant for creating the state space. `etomcrl` is able to translate automatically a big part of the *Erlang* language. We satisfactorily used the non-determinism in μ`CRL` for simulating the users activating our system with all kind of possible requests. Once we have the process algebra specification, we use the μ`CRL toolset` for generating the state space of the system. As the state space can be big, we have to reduce it before model checking. The reduction in the graph and the checking of the formulas is done using `CADP`.

We applied our method successfully to the *VoDKA* system. Following the proposed approach, we managed to extract different kinds of capacity properties about

the system. We could automatically extract information like the number of times that a *Media Object* can be played or how many people can watch MO1 such that the system can still serve MO2; following the same method, we can also extract information about system architecture, system bottlenecks, or even get some hints about how the current architecture would need to be modified when adding a new resource. In general, the results were optimal and we managed to get answers for most of the questions we had stated as goals. However, the approach has some limitations we have described in detail: some of them inherent to the use of model checking (mainly state space explosion), and others related to the kind of properties we are using for extracting information from the system (some performance properties are difficult to formulate as logical expressions over the state space). Apart from the limits of the approach, the tools used have also some limitations, like the time needed for generating the state space or the problems for finding the right reduction rules for making the state space smaller. Trying to overcome that limits and at the same time trying to show that the method is generic and can be implemented using other tools, we applied it again using `McErlang`. This tool allows the direct model checking of *Erlang* programs. We have shown that `McErlang` can be used both for generating a state space similar to the one obtained with the μ`CRL toolset`, and for model checking the properties about the *VoDKA* software architecture in a similar way to `CADP`. `McErlang` lacks some of the features found in mature tools like μ`CRL toolset` and `CADP`, but it has some advantages due to the approach of working directly on the *Erlang* level: it is able to analyze the real state space of the system in detail, accessing all kind of *Erlang* structures of the real processes of the system. This is more powerful than reasoning only about actions and states without information as it is the case with μ`CRL toolset` and `CADP`. We have also described how to use `McErlang` in a creative way for stopping the graph generation in a more flexible way whenever we are in a path where we already know that the next states are not relevant for the property we are trying to check. Doing this, we have been able to extract properties from state spaces that where too big for being generated completely with the previous tools.

Creating a method that could be used by people without expertise in the area of formal methods was also one of the goals of the thesis. On top of all the specialized tools we use, we have developed the prototype for an application that hides the underlying details making easier the extraction of the system properties. The results are very promising and although for advanced properties the users are probably always going to need to know the internal details of the method, we have simplified a lot the extraction of a relevant set of properties and architectural information.

An important thing to remark is that the method proposed is not only flexible in the tools used, but also in the target systems it can be applied to. In this thesis, we have addressed the extraction of properties from the *VoDKA* system, but as we have explained in Chapter 9, the same could be applied to other similar systems, sharing the main features of *VoDKA*, namely: having a flexible architecture, being aware of the underlying hardware resources (or simulating that by some kind of environment), and use of design patterns as the basis for creating the software components.

Chapter 11

Open paths for future research

The results obtained in this thesis open all kind of research paths that would be interesting and challenging to explore. Again, we have already described the main open paths along the different parts of the thesis manuscript, but we will summarize the most relevant ones here.

About the *VoDKA* system itself, there are three lines that are currently the most interesting ones: to develop the capacity of automatically auto-design the system architecture in order to improve its performance; to improve the system architecture by having more and better components; and to increase the code reusability, mainly by using better and more sophisticated design patterns. This is mainly a task for the system architects, but it cannot be easily done without the help of much better tools that the ones available nowadays.

The conclusions of the thesis show that extracting functional and non-functional (e.g. about performance) information automatically from the source code and the system configuration is possible. Main future paths should focus in improving that results in the following four lines:

- Explore the use of other tools from the area of formal methods inside our method. The promising results obtained using `McErlang` suggest that more experiments of using that model checker for *Erlang* in the *VoDKA* system should be carried out. It would be also desirable to compare what we can get from other alternative tools apart from `McErlang`. Also, although we have used some small tools from the area of theorem proving, it would be more than interesting to try to combine them deeper with the model checking approach.

- Complement the study about the limits of the approach and which properties cannot be extracted this way. We have managed to extract interesting results, but it would be interesting to formally explore where we can get with this method and where are the limits that cannot be overcame. For those limits, variations of our method should be proposed.

- Create better top level applications doing exactly what the user wants without removing flexibility and power. We have created a first prototype, but it should be evolved towards a mature and usable application. In order to decide what should be in the interface, a detailed research on the concrete needs of the system architects should be carried out.

- Apply to other systems and learn from that experience. We have shown that our method is not only valid for *VoDKA*, but also for other distributed systems. A natural next step is to apply carefully the method to other systems, what is surely going to produce improvements and changes in the method itself and the underlying tools.

As we said with more detail in the discussion and conclusions of Chapter 9, we want to built upon this research to create a set of innovative tools that can give feedback to software architectures for helping them to increase the quality of their systems.

Part V

Appendixes

In the following appendixes we include some documentation that complements what has been explained in the previous chapters. They are not part of the core research that has been done, but complete or help to understand it in some way.

Appendix A

`etomcrl` tool: simple example, tool usability and other case studies

Contents

A.1 How the tool works: a simple translation example

In this section, a trivial translation from *Erlang* source code to a μ`CRL` specification is given. The trivial example is composed by a simple supervision tree that starts two processes, one for the server (a trivial disk storage implementation) and the other for the users accessing the disk server. Hereafter, the most important parts of the three *Erlang* modules implementing the example and the resulting μ`CRL` specification are shown.

A.1.1 Original *Erlang* source code of the example

A.1.1.1 The supervision tree: `st.erl`

The module `st` implements the call back functions for a very simple supervision tree (the *Erlang* standard behaviour that models a tree of processes supervising their children). The module only exports two functions, one is the `start_link` that is going to be used in order to start the software, and the other the `init` function, which is going to be the call-back function used by the module `supervisor` in order to extract the configuration parameters of the supervision tree to be created. `init` answers with a standard structure where the tree is described: in this case, formed by two *worker* processes supervised by one node. The *Erlang* atoms and numbers are used to configure the policies for re-starting nodes in case of processes crashing, and other similar parameters.

The user would start the program with the call to `st:start_link(MOList)`. Where `MOList` is the (configurable at start up) list of *Media Objects* that are going to be available in the server. Then the function would call to the `start_link` of the supervisor module, which will perform generic initialization steps common to any supervision tree, and will extract the configuration of this tree from the result of the function `init`.

```
-module(st).
-behaviour(supervisor).

-export([start_link/1]).
-export([init/1]).

start_link(MOList) ->
    supervisor:start_link(?MODULE,MOList).

init(MOList) ->
    Disk =
      {disk,{disk,start_link,[MOList]},
                    permanent,2000,worker, []},
    Users =
      {user_simulation,{users,start_link,[disk]},
             permanent,2000,worker,[]},
    {ok,{{one_for_one,0,5000},
         [Disk,Users]}}.
```

A.1.1.2 A simple generic server: `disk.erl`

The worker process implementing the storage containing the list of *Media Objects* uses the `gen_server` behaviour. The module implementing the call-back functions is shown below. Apart from the standard `start_link`, two more functions are exported: the `init`, used by the generic server module for obtaining the initial state; and the `handle_call`, implementing the call-backs for the two messages that can be received from the user. The initial state in this case is the list of *Media Objects* and the number of connections that are in use initially (which is always zero). The messages that can be received are either a lookup or a request for playing a given *Media Object*. If the server receives a lookup, in case the requested

Media Object is present in the disk and there are still available connections, the reply sends an affirmative result (setting the value to 1); in other case it sends a negative result. If the message is a play, the number of used connections is updated. It is interesting to note that a lot of things, including the server loop, are hidden here and only present in the code of the behaviour.

Another interesting thing to note is that even for this small example we are already working with an abstraction of the source code. In a real implementation, when a play message arrives, something more than updating the number of connections should be done. A number of processes should be created and then the actual streaming should be done. Also, when the streaming stops, the connection should be released. In this case we abstract from this and we only would be able to look, while doing verification, to the performance of the system when the load is continuously increasing. A very similar abstraction is done in the more complex source code of the *VoDKA* system.

```
-module(disk).
-behaviour(gen_server).

-define(MAX_CONNECTIONS, 2).

-export([start_link/1]).
-export([init/1, handle_call/3]).

start_link(Movies) -
    gen_server:start_link({local,?MODULE},?MODULE, Movies, []).

init(Movies) ->
    {ok, {Movies,0}}.

handle_call({lookup,MO}, From, {Movies,Connections}) ->
    case lists:member(MO,Movies) and
         (?MAX_CONNECTIONS >= Connections) of
        true ->
            {reply, {lookupAns, 1, MO}, {Movies, Connections}};
        false ->
            {reply, {lookupAns, 0, MO}, {Movies,Connections}}
    end;
handle_call({play, MO, Dest}, From, {Movies, Connections}) ->
    {reply, self(), {Movies, Connections + 1}}.
```

A.1.1.3 A trivial client: `users.erl`

Finally, we show the code of a very simple client. The client is a normal *Erlang* process and it does not implement any behaviour. When it is started, it receives the process identifier of the `disk`, and it creates a new process calling to `spawn_link`, where the function `init` is evaluated. The main loop is now explicit, and it does a request for the *Media Object* $m1$ to the disk device. If the answer is negative, it ignores it and loops, and if it is positive, it requests to play the movie before looping.

```
-module(users).
```

```
-export([start_link/1, init/1]).

start_link(Disk) ->
   {ok, spawn_link(?MODULE, init,[Disk])}.

init(Disk) ->
   loop(Disk).

loop(Disk) ->
    {lookupAns, Answer, MO} =
        gen_server:call(Disk,{lookup,m1}),
    case Answer of
        0 ->
            loop(Disk);
        1 ->
            gen_server:call(Disk,{play,MO,user}),
            loop(Disk)
    end.
```

A.1.2 μCRL specification generated automatically from the example

Using the `etomcrl` tool, presented in Chapter 8, the above supervision tree with two processes can be translated automatically to the following μCRL specification. Instead of showing the whole file, which contains almost a thousand lines of μCRL, we will comment the more interesting parts one by one.

We omit the first part of the specification, which defines in μCRL the rewriting rules for all the data types (*sorts* in μCRL notation) and the pure functions. `Bool`, `Natural`, `Term` (for *Erlang* terms) and all kind of operations are defined over them. An example of the kind of code we find in this first part is the following one, where equality over terms and the if command are partially defined:

```
map
     eq:  Term # Term -> Bool
     if: Bool # Term # Term -> Term
var
     T1,H1,MCRLTerm1,MCRLTerm2: Term
     MCRLBool: Bool
     N: Natural
rew
     if(T,MCRLTerm1,MCRLTerm2) = MCRLTerm1
     if(F,MCRLTerm1,MCRLTerm2) = MCRLTerm2
     eq(pid(N),MCRLTerm2) =
           and(is_pid(MCRLTerm2),eq(N,pid1(MCRLTerm2)))
     eq(MCRLTerm1,pid(N)) =
           and(is_pid(MCRLTerm1),eq(pid1(MCRLTerm1),N))
     eq(int(N),MCRLTerm2) =
           and(is_int(MCRLTerm2),eq(N,int1(MCRLTerm2)))
     eq(MCRLTerm1,int(N)) =
           and(is_int(MCRLTerm1),eq(int1(MCRLTerm1),N))
     ...
```

But the most interesting part is the one defining the actions, the communication pairs, and the processes that interact in the model.

For our example, the following actions are created:

```
act
bufferfull: Term
gen_server_call,gscall,buffercall: Term # Term # Term
gen_server_cast,gscast,buffercast: Term # Term
send,gsinfo,bufferinfo: Term # Term
gshcall,handle_call,call: Term # Term # Term
gshcast,handle_cast,cast: Term # Term
gshinfo,handle_info,info: Term # Term
gen_server_reply,gen_server_replied,reply: Term # Term # Term
```

Some of them model the communication between the artificial buffers that are added as explained by the `etomcrl` tool and the processes, and others model the *Erlang* communications between the different generic servers. In μCRL the actions can have parameters. In our case, the parameters are going to be translated from the *Erlang* terms that appear in the original messages.

But, in order to communicate, the actions need to be placed in pairs. The following code shows how they synchronize:

```
comm
gen_server_call | gscall = buffercall
gen_server_cast | gscast = buffercast
send | gsinfo = bufferinfo
gshcall | handle_call = call
gshcast | handle_cast = cast
gshinfo | handle_info = info
gen_server_reply | gen_server_replied = reply
```

After the actions and communication are specified, we enter into the process part.

The specification of the buffer is shown below. Inside it, we can see that the size of the buffer is limited. If the buffer is full, messages can only be removed from the state. If not, a non-deterministic choice is made in order to decide if a new message (cast, call or info, the three possible messages of a generic server in *Erlang*) is received, or we remove the first from the state. The μCRL code of this implementation is not generated from the *Erlang* source code; instead, the same μCRL code is reused for all the translations.

```
proc
    Server_Buffer(MCRLSelf: Term, Messages: GSBuffer) =
     (bufferfull(MCRLSelf).
      (gshcast(MCRLSelf,cast_term(Messages)).
         Server_Buffer(MCRLSelf,rmhead(Messages))
       <| is_cast(Messages) |>
       (gshinfo(MCRLSelf,info_term(Messages)).
          Server_Buffer(MCRLSelf,rmhead(Messages))
        <| is_info(Messages) |>
        (gshcall(MCRLSelf,call_term(Messages),
                          call_pid(Messages)).
```

```
          Server_Buffer(MCRLSelf,rmhead(Messages))
        <| is_call(Messages) |>
        delta))))
    <| maxbuffer(Messages) |>
    (sum(Msg: Term,
      sum(From: Term,
       gscall(MCRLSelf, Msg, From).
       Server_Buffer(MCRLSelf,
                     addcall(Msg,From,Messages)))) +
     sum(Msg: Term,
         gscast(MCRLSelf, Msg).
         Server_Buffer(MCRLSelf, addcast(Msg,Messages))) +
     sum(Msg: Term,
         gsinfo(MCRLSelf, Msg).
         Server_Buffer(MCRLSelf, addinfo(Msg,Messages))) +
     (gshcast(MCRLSelf,cast_term(Messages)).
        Server_Buffer(MCRLSelf,rmhead(Messages))
      <| is_cast(Messages) |>
      (gshinfo(MCRLSelf,info_term(Messages)).
        Server_Buffer(MCRLSelf,rmhead(Messages))
       <| is_info(Messages) |>
       (gshcall(MCRLSelf,call_term(Messages),
                         call_pid(Messages)).
          Server_Buffer(MCRLSelf,rmhead(Messages))
        <| is_call(Messages) |>
        delta))))
```

For the buffer, and for all the rest of the processes, the translation adds the process identifier as the first parameter. This is useful in order to reuse actions for the synchronization of the different processes.

After the buffer, we can found in the output file the specification of the user process. We see that the supervision tree disappears in the translation. Now the process has a function initializing, very simple, and then the process main loop.

```
users_init(MCRLSelf:Term,Disk:Term) =
     users_loop(MCRLSelf,Disk)

users_loop(MCRLSelf:Term,Disk:Term) =
     gen_server_call(Disk,tuple(lookup, tuplenil(m1)),MCRLSelf).
     sum(Answer: Term,
       sum(MO: Term,
            gen_server_replied(MCRLSelf,tuple(lookupAns,
                           tuple(Answer, tuplenil(MO))),Disk).
            (users_loop(MCRLSelf,Disk)
                 <| eq(equal(Answer,int(0)),true) |>
            (gen_server_call(Disk,tuple(play, tuple(MO,
                                  tuplenil(user))),MCRLSelf).
            sum(MCRLFree0: Term,
                 gen_server_replied(MCRLSelf,MCRLFree0,Disk).
                 users_loop(MCRLSelf,Disk))
            <| eq(equal(Answer,int(s(0))),true) |>
            delta))))
```

The main loop has a very similar structure to the original *Erlang* code. Here we can see that tracing errors back would still be pretty simple. The functions sending messages in the call-back module of the generic server are substituted by actions, and the rest is mainly syntactic translation from *Erlang* to μCRL.

Finally, we can see the code of the disk device:

```
disk_init(MCRLSelf:Term,Movies:Term) =
     disk_serverloop(MCRLSelf,tuple(Movies,
                                tuplenil(int(0))))

disk_serverloop(MCRLSelf:Term,State:Term) =
     sum(From: Term,
       sum(MO: Term,
         handle_call(MCRLSelf,tuple(lookup, tuplenil(MO)),From).
         assertion(equal(size(State),int(s(s(0))))).
         (gen_server_reply(From,tuple(lookupAns, tuple(int(s(0)),
                                      tuplenil(MO))),MCRLSelf).
         disk_serverloop(MCRLSelf,tuple(element(int(s(0)),State),
                      tuplenil(element(int(s(s(0))),State))))
             <| eq(equal(and(member(MO,element(int(s(0)),State)),
                    mcrl_geq(int(s(s(0))),
                    element(int(s(s(0))),State))),true),true) |>
         (gen_server_reply(From,tuple(lookupAns, tuple(int(0),
                                       tuplenil(MO))),MCRLSelf).
         disk_serverloop(MCRLSelf,
                 tuple(element(int(s(0)),State),
                       tuplenil(element(int(s(s(0))),State))))
         <| eq(equal(and(member(MO,element(int(s(0)),State)),
               mcrl_geq(int(s(s(0))),
               element(int(s(s(0))),State))),false),true) |>
         delta)))) +
     sum(From: Term,
          sum(MO: Term,
               sum(Dest: Term,
                    handle_call(MCRLSelf,tuple(play,
                              tuple(MO, tuplenil(Dest))),From).
                    assertion(equal(size(State),int(s(s(0))))).
                    gen_server_reply(From,MCRLSelf,MCRLSelf).
                    disk_serverloop(MCRLSelf,
                     tuple(element(int(s(0)),State),
                       tuplenil(mcrl_plus(element(int(s(s(0))),
                             State),int(s(0)))))))))
```

It is a bit more complex but also quite similar to the original *Erlang* code. The two `handle_call` clauses are translated into a non-deterministic choice between two actions. Some assertions are added because of the conditions created by the pattern matching. The state is extracted from the code and made now explicit in the server loop (remember that it was hidden in the behaviour-based implementation).

The last part of the μCRL specification is what is called the initialization area. It starts all the processes in parallel with their initial parameters. This part can be seen as a substitution of the supervision module and its call-back module with the concrete configuration for our example.

```
init
     hide({conftau,buffercall,buffercast,bufferinfo},
     encap({handle_call,gen_server_call,
            handle_cast,gen_server_cast,
              gscall,gshcall,gscast,gshcast,send,gsinfo,
              gen_server_reply,gen_server_replied},

           users_init(pid(0),disk) ||
           hide({push_callstack,pop_callstack},
     encap({rcallvalue,wcallvalue,rcallresult,wcallresult},
          CallStack(empty) ||
        disk_init(disk,cons(m1,cons(m2,nil))))) ||
        Server_Buffer(disk,emptybuffer)
          ))
```

The actions that are not relevant for verification purposes are encapsulated and hidden. The goal is to reduce as much as possible the information present in the state space that would be derived from the specification.

A.2 Using the `etomcrl` tool

One of the goals of the `etomcrl` tool was, since the very beginning, being easy to use and available for the research community. Because of this, it was early published as free software in one of the main public repositories. Besides, the tool is easy to compile and install, only needing to follow the very standard techniques used in most of the *Erlang* projects. As an example of this concept, we include in this section some ideas about how the tool is used.

The tool API is defined in the `etomcrl.erl` module. This module acts as the interface between the user and the `etomcrl` tool. It contains high level functions that allow to use the compiler inside the *Erlang* shell or from the command line of any other shell. The main API functions are the following ones:

- The `script/0` function is used in order to execute the `supervisor` function without starting the *Erlang* shell. It obtains the arguments from the string given as a parameter after the `-start` option, and could be used as follows:

  ```
  erl -noshell -s etomcrl script \\
              -start "mod:function([arguments])"
  ```

- The functions `batch/0` and `batch/2` allow to check a list of properties against a list of configurations. To do so, they call the μ`CRL` tools needed to generate the LTS and the CADP tools to model check the properties. Therefore, this functions integrate the `etomcrl tool` with external software. An example of use would be the following one:

  ```
  erl -noshell -s etomcrl batch -file "file1" -property "file2"
  ```

  `file1` should contain a list of `"mod:function([arguments])"`. Those are the programs one wants to verify; `file2` should contain a list of properties expressed in the alternation-free mu-calculus used by the CADP toolset.

`batch/0` takes the command line arguments, reads the contents of the given files and calls `batch/2` which would perform all the other operations.

- `supervisor/3` and `supervisor/6` are the main functions in the tool API. They start all the compilation process. The first one is used if no source and destination directories are specified, and in that case the current directory is taken by default as source directory and for the destination directory the directory specified in the `etomcrl.h` file.

```
supervisor(SrcDir::string(), DestDir::string(),
     Mod::atom(), Fun::atom(), Args::[term()],
     Options::options()) -> string()
```

`ScrDir` is the directory where the source code is located. `Destdir` is the directory where the resulting file with μ`CRL` specification is located. The file is named after the name of the module given as a third argument for the `supervisor/6` function and it has a μ`CRL` extension. `Mod` is an atom representing the name of the module. This module contains the implementation of the root of the supervision tree. `Fun` is the function that starts the supervision tree `Args` is the list of the arguments that are passed to `Fun`. `Options` is a list of tuples with options for the compiler. Supported options are: {`file,Directory`} to store intermediate files, that are created at different stages during the compilation. {`buffer,N`} to set the size of buffer to `int N`. This buffer is used to implement the asynchronous communication between processes in μ`CRL`. This is only used in the last phase of the compilation, when the *Erlang* program is fully translated to μ`CRL`. `N` could also be the *Erlang* atom `none`, meaning synchronous communication.

The function returns the name of the file where the μ`CRL` spec is written.

- The function `instantiator/3` works as a script that generates the μ`CRL` file from the *Erlang* source files and then calls the μ`CRL` tools to generate the corresponding LTS. It has the following specification:

```
instantiator(Mod, Fun, Args) -> term()
```

A.3 Other case study: ATM switch

The `etomcrl` tool has been successfully used in two industrial case studies, one is the *VoDKA* server, subject of this thesis, and the other is introduced in the current section. Both teams working in both case studies have shared their efforts and conclusions during the last years, and some of the tools and steps have been also born out of that collaboration.

A.3.1 An ATM switch Locker

Below, we summarize the main ideas behind the research carried out for the verification of an ATM switch locker using `etomcrl`. More details can be found in [ABD04].

A.3.1.1 Project description

The telecommunication company Ericsson is using the functional programming language *Erlang* for the development of concurrent/distributed software for telecommunications equipment. One of the larger examples of such a system is the AXD 301 high capacity ATM switch [BR98b], used to implement, for example, the backbone network in the UK. The software of this switch consists of about half a million lines of *Erlang* code.

This code is written in a development process that is rather similar to the eXtreme Programming approach: designers write and test it themselves and in small iterations, features are added to the code until a final release stage is reached.

As it was the case of *VoDKA*, in Ericsson the software for large projects like the AXD 301 switch is written according to rather strict design principles. For the AXD, a number of software components are used which have been specified for use in a number of Ericsson projects. These components can be seen as higher-order functions for which certain functions have to be given to determine the specific functionality of the component. About eighty percent of the software implements code for this specific functionality of one of these components, the majority of this for the *generic server* component.

The development process and the use of these library components both ensure that the code is tested many times before the final implementation. For example, during development the software is often written during day-time and tested overnight. The test cases are written by the designers in parallel with the code and a test server automatically runs these test cases.

However, despite this extensive testing, for critical hardware such as telecommunications switches it is clearly preferably to have even higher levels of assurance that the code is correct. The aim, therefore, was to build a formal verification tool that fit into this development process. `etomcrl` tool would be part of this framework.

A.3.1.2 Results of using the tool within this project

The use of the tool is similar to the one presented for *VoDKA* in this thesis, although some of the steps in the method and the kind of properties that are extracted are completely different.

The idea followed in this case study consists of the following steps. The *Erlang* code for a component is automatically translated to a process algebraic specification written in μ`CRL`. We then generate a labelled transition system (LTS) from this μ`CRL` specification by using components of the μ`CRL` toolset. The properties of interest are then written in the logic of the model checker we use, here we use the regular alternation-free μ`-Calculus` to express non-starvation and mutual exclusion. The labelled transition system is then checked against this property using the CÆSAR/ALDÉBARAN toolset. For some properties it is necessary to transform the LTS (e.g., using hiding for non-starvation) so that we can model check with a simpler formulation of the property of interest (e.g., one without alternating fixed points).

In general, the approach uses similar tools and steps, but has different kinds of goals. The properties in this case are the classical ones, trying to find errors in

the software. The effort, therefore, is complementary to the one presented in this thesis.

Appendix B

An example of using `etomcrl` in *VoDKA* with source code

Contents

Hereafter we include, for completeness and ilustrative reasons, the main parts of the source code of one of the *Erlang* models of *VoDKA* that we have presented in Chapter 9, together with the resulting μ`CRL` translation.

B.1 Supervision tree: `vodka.erl`

The following code implements the call-back module for the main supervision tree in the *VoDKA* system: the one that starts the storage and streaming server, together with the users process.

```
-module(vodka).
-behaviour(supervisor).

-export([start_link/1]).
-export([init/1]).

start_link(DeviceList) ->
    supervisor:start_link(?MODULE,DeviceList).

init(DeviceList) ->
    StorageSup =
      {storage_supervisor,{storage,start_link,
                           [DeviceList]},
```

```
                          permanent,2000,supervisor,[]},
    StorageSched =
      {storage_sched,{storage_sched,start_link,[storage_group]},
                     permanent,2000,worker,[]},
    StreamingSched =
      {streaming_sched,{streaming_sched,start_link,
                                              [storage_sched]},
                     permanent,2000,worker, []},
    Users =
      {user_simulation,{users,start_link,[streaming_sched]},
             permanent,2000,worker,[]},
    {ok,{{one_for_one,0,5000},
         [Users,StreamingSched,StorageSched,StorageSup]}}.
```

B.2 Supervision tree: `storage.erl`

The code below implements the storage level, which is also a supervision subtree of the general one. It starts the `storage_group` and the list of storage devices.

```
-module(storage).
-behaviour(supervisor).

-export([start_link/1]).
-export([init/1]).

start_link(Drivers) ->
    supervisor:start_link(?MODULE, Drivers).

init(Drivers) ->
    StorageGroup =
      {storage_group,{storage_group,start_link,
                      [lists:map(fun({DriverName,_,_,_,_,_}) ->
                                    DriverName
                                 end,Drivers)]
     },
                     permanent,2000,worker,[]},
    DriverSpecs =
      lists:map(fun({DriverName, DriverType, MOs,
                          MaxConnections, MaxBandwidth, Cost}) ->
{DriverName,
 {DriverType,start_link,
                            [DriverName, MOs,
                             MaxConnections,MaxBandwidth, Cost]},
 permanent,2000,worker,[]}
                end,Drivers),
    {ok,{{one_for_one,2,5000}, [StorageGroup|DriverSpecs]}}.
```

B.3 Generic server: `storage_sched.erl`

The storage level acts as a middleware between the storage devices and the streaming level. It forwards messages both ways doing some scheduling in the middle if needed.

```
-module(storage_sched).
-behaviour(gen_server).

-export([start_link/1]).
-export([init/1, handle_call/3]).

start_link(StorageGroup) ->
    gen_server:start_link({local,?MODULE},?MODULE,
                          StorageGroup, []).

init(StorageGroup) ->
    {ok, {StorageGroup,0, 0}}.

handle_call({lookup,MO,Profile}, From,
           {StorageGroup,UsedConnections, UsedBandwidth}) ->
    {lookupAns,Options} =
           gen_server:call(StorageGroup,{lookup,MO,Profile}),
    {reply, {lookupAns,cost(Options,UsedConnections)},
     {StorageGroup,UsedConnections, UsedBandwidth}};

handle_call({play,MO, Bw, Pid}, From,
           {StorageGroup, UsedConnections, UsedBandwidth}) ->
    Reply = gen_server:call(StorageGroup, {play,MO, Bw, Pid}),
    {reply, Reply,
           {StorageGroup, UsedConnections+1, UsedBandwidth+Bw}}.

cost(Options,Connections) ->
   Options.
```

B.4 Generic server: storage_group.erl

This generic server groups all the storage devices, acting as a unique proxy for the upper levels in the system configuration. It forwards messages to all the connected devices and then groups the answers back.

```
-module(storage_group).
-behaviour(gen_server).

-export([start_link/1]).
-export([init/1, handle_call/3]).

start_link(StorageDrivers) ->
    gen_server:start_link({local,?MODULE},?MODULE,
                          StorageDrivers, []).

init(StorageDrivers) ->
    {ok, {StorageDrivers,0, 0}}.

handle_call({lookup,MO,Profile}, From,
         {StorageDrivers,UsedConnections, UsedBandwidth}) ->
    Options = send_all({lookup,MO,Profile},StorageDrivers,[]),
    {reply, {lookupAns,cost(Options,UsedConnections)},
     {StorageDrivers,UsedConnections, UsedBandwidth}};
```

```
handle_call({play, MO, Bw, Pid}, From,
         {StorageDrivers, UsedConnections, UsedBandwidth}) ->
    Reply = gen_server:call(Pid, {play,MO, Bw}),
    {reply, Reply,
         {StorageDrivers, UsedConnections+1, UsedBandwidth+Bw}}.

send_all(Message,[], Options) -> Options;
send_all(Message,[Driver|Drivers],Options) ->
   {lookupAns, NewOptions} = gen_server:call(Driver,Message),
   send_all(Message, Drivers, NewOptions++Options).

cost(Options, Connections) ->
  Options.
```

B.5 Generic server: streaming_sched.erl

The streaming level acts as a middleware between the storage and the user. It forwards messages both ways doing some scheduling in the middle.

```
-module(streaming_sched).
-behaviour(gen_server).

-export([start_link/1]).
-export([init/1, handle_call/3]).

start_link(StorageSched) ->
    gen_server:start_link({local,?MODULE},?MODULE,
                          StorageSched, []).

init(StorageSched) ->
    {ok, {StorageSched, 0, 0}}.

handle_call({lookup,MO,Profile}, From,
              {StorageSched,UsedConnections, UsedBandwidth}) ->
    {lookupAns,Options} =
              gen_server:call(StorageSched,{lookup,MO,Profile}),
    {reply, {lookupAns, MO, Profile, cost(Options,UsedConnections)},
     {StorageSched,UsedConnections, UsedBandwidth}};

handle_call({play,MO, Bw, Pid}, From,
               {StorageSched, UsedConnections, UsedBandwidth}) ->
    Reply = gen_server:call(StorageSched, {play,MO, Bw, Pid}),
    {reply, Reply,
              {StorageSched, UsedConnections+1, UsedBandwidth+Bw}}.

cost(Options,Connections) ->
   Options.
```

B.6 Generic server: disk_storage.erl

The code for the storage devices is quite similar. They store a list of *Media Objects* and they react to the messages in the protocol depending on the availability of media an resources. The differences between then are more related to the underlying resources they are managing. We only include here the code for the disk.

However, in the translation we include all of them (tape, disk and cd), to show their similarities also in the μCRL specification.

```
-module(disk_storage).
-behaviour(gen_server).

-export([start_link/5]).
-export([init/1, handle_call/3]).

start_link(Name, Movies, MaxConnections, MaxBandwidth, Cost) ->
    gen_server:start_link({local,Name}, ?MODULE,
         [Movies, MaxConnections, MaxBandwidth, Cost], []).

init([Movies, MaxConnections, MaxBandwith, Cost]) ->
    {ok, {Movies,MaxConnections,MaxBandwith,Cost,0,0}}.

handle_call({lookup,MO,Profile}, From,
    {Movies, MaxConnections, MaxBandwidth,
                              Cost, Connections, UsedBandwidth}) ->
    {reply,
     {lookupAns,
      scheduling(MO, Profile, Movies,
 UsedBandwidth, Connections,
                 MaxConnections, MaxBandwidth,
 Cost, self())},
     {Movies, MaxConnections, MaxBandwidth,
      Cost, Connections, UsedBandwidth}};

handle_call({play, MO, Bandwidth}, From,
    {Movies, MaxConnections, MaxBandwidth,
             Cost, Connections, UsedBandwidth}) ->
    {reply, self(), {Movies, MaxConnections, MaxBandwidth, Cost,
     Connections + 1, UsedBandwidth + Bandwidth}}.

scheduling(MO, Profile, Movies, UsedBandwidth, UsedConnections,
   MaxConnections, MaxBandwidth, Cost, ProcId) ->
   case UsedConnections >= MaxConnections of
     true ->
         [];
     false ->
   case UsedBandwidth >= MaxBandwidth of
       true ->
   [];
       false ->
   AvailableBw =
                        check_available_bw(MO, Profile, Movies),
   scheduling_still_resources(
```

```
                        AvailableBw, UsedBandwidth, UsedConnections,
       MaxConnections, MaxBandwidth, Cost, ProcId)
   end
   end.

scheduling_still_resources ([], UsedBandwidth, UsedConnections,
           MaxConnections, MaxBandwidth, Cost, ProcId) ->
    [];
scheduling_still_resources ([B|RestBandwidths],
                UsedBandwidth, UsedConnections,
        MaxConnections, MaxBandwidth, Cost, ProcId) ->
    case MaxBandwidth >= (UsedBandwidth+B) of
false ->
    scheduling_still_resources (
                    RestBandwidths, UsedBandwidth, UsedConnections,
            MaxConnections, MaxBandwidth, Cost, ProcId);
true -> [
  {B,
   cost(B,UsedBandwidth, UsedConnections,
                         MaxConnections, MaxBandwidth, Cost),
   ProcId}
  |
  scheduling_still_resources (
                    RestBandwidths, UsedBandwidth, UsedConnections,
                    MaxConnections, MaxBandwidth, Cost, ProcId)
 ]
    end.

cost(MOBandwidth, UsedBandwidth, UsedConnections,
     MaxConnections, MaxBandwidth, Cost) ->
     MOBandwidth * (UsedConnections+1) * Cost.

check_available_bw (MO, Profile, []) ->
    [];
check_available_bw (MO1, Profile,
                   [{MO2,AvailableProfile}|OtherMOs]) ->
    case (MO1 == MO2) of
true ->
    list_intersection (Profile, AvailableProfile);
false ->
    check_available_bw (MO1, Profile, OtherMOs)
    end.

list_intersection ([],List) ->
    [];
list_intersection (List, []) ->
    [];
list_intersection ([Hd1|Tail1], [Hd2|Tail2]) ->
    case Hd1 >= Hd2 of
        false ->
            list_intersection (Tail1, [Hd2|Tail2]);
        true ->
            case Hd1 == Hd2 of
```

```
                true ->
                  [Hd1 | list_intersection (Tail1, Tail2)];
                false ->
                  list_intersection ([Hd1|Tail1], Tail2)
            end
    end.
```

B.7 μCRL code for the main part of the example

The transformation for the input *Erlang* code shown above, after removing the innitial part of the μCRL file (were the data types and the pure rewriting rules are described), would be as follows:

```
proc

cd_storage_serverloop(MCRLSelf:Term,State:Term) =
  sum(From: Term,
       sum(MO: Term,
            sum(Profile: Term,
                handle_call(MCRLSelf,tuple(lookup,
                           tuple(MO, tuplenil(Profile))),From).
                assertion(equal(size(State),
                           int(s(s(s(s(s(s(0))))))))).
                gen_server_reply(From,tuple(lookupAns,
                  tuplenil(cd_storage_scheduling(MO,Profile,
                    element(int(s(0)),State),
                     element(int(s(s(s(s(s(s(0)))))),State),
                      element(int(s(s(s(s(s(0))))),State),
                       element(int(s(s(s(0)))),State),
                        element(int(s(s(s(0)))),State),
                         element(int(s(s(s(s(0))))),State),
                          MCRLSelf))),MCRLSelf).
                 cd_storage_serverloop(MCRLSelf,
                               tuple(element(int(s(0)),State),
                 tuple(element(int(s(s(0))),State),
                 tuple(element(int(s(s(s(0)))),State),
                 tuple(element(int(s(s(s(s(0))))),State),
                 tuple(element(int(s(s(s(s(s(0)))))),State),
                 tuplenil(element(int(s(s(s(s(s(s(0)))))),
                                          State))))))))))
                 +
  sum(From: Term,
       sum(MO: Term,
            sum(Bandwidth: Term,
                 handle_call(MCRLSelf,tuple(play,
                           tuple(MO, tuplenil(Bandwidth))),From).
                 assertion(equal(size(State),
                                    int(s(s(s(s(s(s(0))))))))).
                 gen_server_reply(From,MCRLSelf,MCRLSelf).
                 cd_storage_serverloop(MCRLSelf,
                                tuple(element(int(s(0)),State),
                 tuple(element(int(s(s(0))),State),
                 tuple(element(int(s(s(s(0)))),State),
```

```
                tuple(element(int(s(s(s(s(0))))),State),
                tuple(plus(element(int(s(s(s(s(s(0))))))),State),
                                                  int(s(0))),
                tuplenil(plus(element(int(s(s(s(s(s(0)))))),
                                  State),Bandwidth))))))))))

cd_storage_init(MCRLSelf:Term,MCRLArg1:Term) =
     (cd_storage_serverloop(MCRLSelf,tuple(hd(MCRLArg1),
                tuple(hd(tl(MCRLArg1)),
                     tuple(hd(tl(tl(MCRLArg1))),
                          tuple(hd(tl(tl(tl(MCRLArg1)))),
                               tuple(int(0),
                                     tuplenil(int(0))))))))
      <| eq(equal(tl(tl(tl(tl(MCRLArg1)))),nil),true) |> delta)

tape_storage_serverloop(MCRLSelf:Term,State:Term) =
     sum(From: Term,
          sum(MO: Term,
               sum(Profile: Term,
                    handle_call(MCRLSelf,tuple(lookup,
                           tuple(MO, tuplenil(Profile))),From).
                    assertion(equal(size(State),
                           int(s(s(s(s(s(0))))))))).
                    gen_server_reply(From,tuple(lookupAns,
                     tuplenil(tape_storage_scheduling(MO,Profile,
                        element(int(s(0)),State),
                         element(int(s(s(s(s(s(s(0)))))),State),
                          element(int(s(s(s(s(s(0))))),State),
                           element(int(s(s(s(0)))),State),
                            element(int(s(s(s(0)))),State),
                             element(int(s(s(s(s(0))))),State),
                             MCRLSelf))),MCRLSelf).
                    tape_storage_serverloop(MCRLSelf,
                     tuple(element(int(s(0)),State),
                      tuple(element(int(s(s(0))),State),
                       tuple(element(int(s(s(s(0)))),State),
                       tuple(element(int(s(s(s(s(0))))),State),
                       tuple(element(int(s(s(s(s(s(0)))))),State),
                       tuplenil(element(int(s(s(s(s(s(0)))))),
                       State))))))))) +
     sum(From: Term,
          sum(MO: Term,
               sum(Bandwidth: Term,
                    handle_call(MCRLSelf,tuple(play,
                            tuple(MO, tuplenil(Bandwidth))),From).
                    assertion(equal(size(State),
                            int(s(s(s(s(s(0))))))))).
                    gen_server_reply(From,MCRLSelf,MCRLSelf).
                    tape_storage_serverloop(MCRLSelf,
                                  tuple(element(int(s(0)),State),
                    tuple(element(int(s(s(0))),State),
                    tuple(element(int(s(s(s(0)))),State),
                    tuple(element(int(s(s(s(s(0))))),State),
```

```
                    tuple(plus(element(int(s(s(s(s(s(0))))))),State),
                                       int(s(0))),
                    tuplenil(plus(element(int(s(s(s(s(s(s(0)))))))),
                                       State),
                      Bandwidth))))))))))

tape_storage_init(MCRLSelf:Term,MCRLArg1:Term) =
     (tape_storage_serverloop(MCRLSelf,
          tuple(hd(MCRLArg1), tuple(hd(tl(MCRLArg1)),
             tuple(hd(tl(tl(MCRLArg1))),
                tuple(hd(tl(tl(tl(MCRLArg1)))),
                   tuple(int(0), tuplenil(int(0))))))))
      <| eq(equal(tl(tl(tl(tl(MCRLArg1)))),nil),true) |> delta)

storage_group_serverloop(MCRLSelf:Term,State:Term) =
     sum(From: Term,
          sum(MO: Term,
           sum(Profile: Term,
                handle_call(MCRLSelf,tuple(lookup,
                      tuple(MO, tuplenil(Profile))),From).
                assertion(equal(size(State),int(s(s(s(0))))))).
                storage_group_send_all(MCRLSelf,
                      tuple(lookup, tuple(MO,
                       tuplenil(Profile))),
                         element(int(s(0)),State),nil).
                sum(Options: Term,
                     rcallresult(MCRLSelf,Options).
                     gen_server_reply(From,
                       tuple(lookupAns,
                       tuplenil(storage_group_cost(Options,
                       element(int(s(s(0))),State)))),MCRLSelf).
                     storage_group_serverloop(MCRLSelf,
                       tuple(element(int(s(0)),State),
                       tuple(element(int(s(s(0))),State),
                       tuplenil(element(int(s(s(s(0)))),
                         State))))))))) +
     sum(From: Term,
          sum(MO: Term,
               sum(Bw: Term,
                    sum(Pid: Term,
                         handle_call(MCRLSelf,tuple(play,
                              tuple(MO, tuple(Bw,
                                   tuplenil(Pid)))),From).
                         assertion(equal(size(State),
                                        int(s(s(s(0)))))).
                         gen_server_call(Pid,tuple(play,
                                 tuple(MO, tuplenil(Bw))),
                                                MCRLSelf).
                         sum(Reply: Term,
                          gen_server_replied(MCRLSelf,Reply,Pid).
                          gen_server_reply(From,Reply,MCRLSelf).
                          storage_group_serverloop(MCRLSelf,
                             tuple(element(int(s(0)),State),
```

```
                              tuple(plus(element(
                               int(s(s(0))),State),int(s(0))),
                              tuplenil(plus(element(
                                       int(s(s(s(0)))),State),
                                     Bw))))))))))

storage_group_send_all(MCRLSelf:Term,Message:Term,
                               MCRLArg1:Term,Options:Term) =
     (wcallresult(MCRLSelf,Options)
      <| eq(equal(MCRLArg1,nil),true) |>
               gen_server_call(hd(MCRLArg1),Message,MCRLSelf).
     sum(NewOptions: Term,
          gen_server_replied(MCRLSelf,tuple(lookupAns,
                      tuplenil(NewOptions)),hd(MCRLArg1)).
          storage_group_send_all(MCRLSelf,Message,
                      tl(MCRLArg1),append(NewOptions,Options))))

storage_group_init(MCRLSelf:Term,StorageDrivers:Term) =
     storage_group_serverloop(MCRLSelf,
          tuple(StorageDrivers, tuple(int(0), tuplenil(int(0)))))

storage_sched_serverloop(MCRLSelf:Term,State:Term) =
     sum(From: Term,
          sum(MO: Term,
               sum(Profile: Term,
                    handle_call(MCRLSelf,tuple(lookup,
                                tuple(MO, tuplenil(Profile))),From).
                    assertion(equal(size(State),int(s(s(s(0))))))).
                    gen_server_call(element(int(s(0)),State),
                              tuple(lookup, tuple(MO,
                                   tuplenil(Profile))),MCRLSelf).
                    sum(Options: Term,
                     gen_server_replied(MCRLSelf,
                        tuple(lookupAns, tuplenil(Options)),
                                  element(int(s(0)),State)).
                     gen_server_reply(From,
                       tuple(lookupAns,
                         tuplenil(storage_sched_cost(Options,
                         element(int(s(s(0))),State)))),MCRLSelf).
                     storage_sched_serverloop(MCRLSelf,tuple(
                        element(int(s(0)),State),
                           tuple(element(int(s(s(0))),State),
                            tuplenil(element(int(s(s(s(0)))),
                              State)))))))) +
     sum(From: Term,
          sum(MO: Term,
               sum(Bw: Term,
                  sum(Pid: Term,
                   handle_call(MCRLSelf,tuple(play,
                    tuple(MO, tuple(Bw, tuplenil(Pid)))),From).
                    assertion(equal(size(State),int(s(s(s(0))))))).
                   gen_server_call(element(int(s(0)),State),
                  tuple(play, tuple(MO, tuple(Bw, tuplenil(Pid)))),
```

```
                                  MCRLSelf).
                sum(Reply: Term,
                 gen_server_replied(MCRLSelf,Reply,
                                 element(int(s(0)),State)).
                 gen_server_reply(From,Reply,MCRLSelf).
                 storage_sched_serverloop(MCRLSelf,
                          tuple(element(int(s(0)),State),
                 tuple(plus(element(int(s(s(0))),State),
                                                  int(s(0))),
                 tuplenil(plus(element(int(s(s(s(0)))),
                                                State),
                                           Bw)))))))))


storage_sched_init(MCRLSelf:Term,StorageGroup:Term) =
     storage_sched_serverloop(MCRLSelf,
           tuple(StorageGroup, tuple(int(0), tuplenil(int(0)))))


streaming_sched_serverloop(MCRLSelf:Term,State:Term) =
     sum(From: Term,
          sum(MO: Term,
               sum(Profile: Term,
                    handle_call(MCRLSelf,tuple(lookup,
                              tuple(MO, tuplenil(Profile))),From).
                    assertion(equal(size(State),int(s(s(s(0))))))).
                    gen_server_call(element(int(s(0)),State),
                     tuple(lookup, tuple(MO,
                              tuplenil(Profile))),MCRLSelf).
                    sum(Options: Term,
                         gen_server_replied(MCRLSelf,
                           tuple(lookupAns, tuplenil(Options)),
                                  element(int(s(0)),State)).
                         gen_server_reply(From,
                           tuple(lookupAns,
                             tuple(MO, tuple(Profile,
                               tuplenil(streaming_sched_cost(
                                  Options,element(int(s(s(0))),
                                    State))))))),MCRLSelf).
                         streaming_sched_serverloop(MCRLSelf,
                            tuple(element(int(s(0)),State),
                            tuple(element(int(s(s(0))),State),
                            tuplenil(
                            element(int(s(s(s(0)))),
                             State)))))))) +
     sum(From: Term,
      sum(MO: Term,
       sum(Bw: Term,
         sum(Pid: Term,
          handle_call(MCRLSelf,tuple(play,
                      tuple(MO, tuple(Bw,
                      tuplenil(Pid)))),From).
          assertion(equal(size(State),int(s(s(s(0))))))).
```

```
          gen_server_call(element(int(s(0)),State),
                tuple(play, tuple(MO,
                tuple(Bw, tuplenil(Pid)))),MCRLSelf).
          sum(Reply: Term,
            gen_server_replied(MCRLSelf,Reply,
                 element(int(s(0)),State)).
            gen_server_reply(From,Reply,MCRLSelf).
              streaming_sched_serverloop(MCRLSelf,
               tuple(element(int(s(0)),State),
               tuple(plus(element(int(s(s(0))),State),
                                              int(s(0))),
               tuplenil(plus(element(int(s(s(s(0)))),State),
                             Bw)))))))))

streaming_sched_init(MCRLSelf:Term,StorageSched:Term) =
     streaming_sched_serverloop(MCRLSelf,
        tuple(StorageSched, tuple(int(0), tuplenil(int(0)))))

users_init(MCRLSelf:Term,StreamingScheduler:Term) =
     users_loop(MCRLSelf,StreamingScheduler)

users_loop(MCRLSelf:Term,StreamingScheduler:Term) =
     (gen_server_call(StreamingScheduler,tuple(lookup, tuple(m2,
     tuplenil(cons(int(s(0)),nil)))),MCRLSelf) +
     gen_server_call(StreamingScheduler,tuple(lookup, tuple(m2,
     tuplenil(cons(int(s(s(0))),nil)))),MCRLSelf) +
     gen_server_call(StreamingScheduler,tuple(lookup, tuple(m1,
     tuplenil(cons(int(s(0)),nil)))),MCRLSelf) +
     gen_server_call(StreamingScheduler,tuple(lookup, tuple(m1,
     tuplenil(cons(int(s(s(0))),nil)))),MCRLSelf) +
     gen_server_call(StreamingScheduler,tuple(lookup, tuple(m3,
     tuplenil(cons(int(s(0)),nil)))),MCRLSelf) +
     gen_server_call(StreamingScheduler,tuple(lookup, tuple(m3,
     tuplenil(cons(int(s(s(0))),nil)))),MCRLSelf)).
     sum(MO: Term,
       sum(Profile: Term,
         sum(Options: Term,
             gen_server_replied(MCRLSelf,tuple(lookupAns,
              tuple(MO, tuple(Profile,
                tuplenil(Options)))),StreamingScheduler).
                (users_loop(MCRLSelf,StreamingScheduler)
                   <| eq(users_choose(Options),fail) |>
              gen_server_call(StreamingScheduler,tuple(play,
                tuple(MO,
                tuple(element(int(s(s(0))),users_choose(Options)),
                tuplenil(element(int(s(0)),
                              users_choose(Options))))))),MCRLSelf).
              sum(MCRLFree0: Term,
                  gen_server_replied(MCRLSelf,
                                  MCRLFree0,StreamingScheduler).
                  users_loop(MCRLSelf,StreamingScheduler))))))

proc streaming_sched_buffer(MCRLSelf: Term, Messages: Buffer) =
```

```
     (bufferfull(MCRLSelf).
      sum(Msg: Term,
          gshcast(MCRLSelf,Msg).
          streaming_sched_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              streaming_sched_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))
     <| maxbuffer(Messages) |>
     sum(Msg: Term,
         sum(From: Term,
             gscall(MCRLSelf, Msg, From).
             streaming_sched_buffer(MCRLSelf,
                           add(gscall(Msg,From),Messages)))
         +
         gscast(MCRLSelf,Msg).
         streaming_sched_buffer(MCRLSelf, add(gscast(Msg),Messages))
         +
         (gshcast(MCRLSelf,Msg).
          streaming_sched_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              streaming_sched_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))

proc storage_sched_buffer(MCRLSelf: Term, Messages: Buffer) =
     (bufferfull(MCRLSelf).
      sum(Msg: Term,
          gshcast(MCRLSelf,Msg).
          storage_sched_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              storage_sched_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))
     <| maxbuffer(Messages) |>
     sum(Msg: Term,
         sum(From: Term,
             gscall(MCRLSelf, Msg, From).
             storage_sched_buffer(MCRLSelf,
                       add(gscall(Msg,From),Messages)))
         +
         gscast(MCRLSelf,Msg).
         storage_sched_buffer(MCRLSelf, add(gscast(Msg),Messages))
         +
         (gshcast(MCRLSelf,Msg).
          storage_sched_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
```

```
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              storage_sched_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))

proc storage_group_buffer(MCRLSelf: Term, Messages: Buffer) =
     (bufferfull(MCRLSelf).
      sum(Msg: Term,
          gshcast(MCRLSelf,Msg).
          storage_group_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              storage_group_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))
     <| maxbuffer(Messages) |>
     sum(Msg: Term,
         sum(From: Term,
             gscall(MCRLSelf, Msg, From).
             storage_group_buffer(MCRLSelf,
                      add(gscall(Msg,From),Messages)))
         +
         gscast(MCRLSelf,Msg).
         storage_group_buffer(MCRLSelf,
                              add(gscast(Msg),Messages))
         +
         (gshcast(MCRLSelf,Msg).
          storage_group_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              storage_group_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))

proc tape_storage_buffer(MCRLSelf: Term, Messages: Buffer) =
     (bufferfull(MCRLSelf).
      sum(Msg: Term,
          gshcast(MCRLSelf,Msg).
          tape_storage_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              tape_storage_buffer(MCRLSelf,
                    removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))
     <| maxbuffer(Messages) |>
     sum(Msg: Term,
         sum(From: Term,
             gscall(MCRLSelf, Msg, From).
```

```
            tape_storage_buffer(MCRLSelf,
                       add(gscall(Msg,From),Messages)))
        +
        gscast(MCRLSelf,Msg).
        tape_storage_buffer(MCRLSelf,
                       add(gscast(Msg),Messages))
        +
        (gshcast(MCRLSelf,Msg).
         tape_storage_buffer(MCRLSelf,removehead(Messages))
         <| ishead(gscast(Msg),Messages) |>
         sum(From: Term,
             gshcall(MCRLSelf,Msg,From).
             tape_storage_buffer(MCRLSelf,removehead(Messages))
             <| ishead(gscall(Msg,From),Messages) |>
             delta)))

proc cd_storage_buffer(MCRLSelf: Term, Messages: Buffer) =
     (bufferfull(MCRLSelf).
      sum(Msg: Term,
          gshcast(MCRLSelf,Msg).
          cd_storage_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              cd_storage_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))
     <| maxbuffer(Messages) |>
     sum(Msg: Term,
         sum(From: Term,
             gscall(MCRLSelf, Msg, From).
             cd_storage_buffer(MCRLSelf,
                       add(gscall(Msg,From),Messages)))
         +
         gscast(MCRLSelf,Msg).
         cd_storage_buffer(MCRLSelf, add(gscast(Msg),Messages))
         +
         (gshcast(MCRLSelf,Msg).
          cd_storage_buffer(MCRLSelf,removehead(Messages))
          <| ishead(gscast(Msg),Messages) |>
          sum(From: Term,
              gshcall(MCRLSelf,Msg,From).
              cd_storage_buffer(MCRLSelf,removehead(Messages))
              <| ishead(gscall(Msg,From),Messages) |>
              delta)))

proc callreturn(Calls: CallStacks) =
     sum(Pid: Term,
        sum(Value:Term,
            (rcallvalue(Pid,Value).
             callreturn(push(Pid,Value,Calls)))
            +
            (wcallvalue(Pid,Value).
```

```
            callreturn(pop(Pid,Calls))
            <| stacked(Pid,Value,Calls) |>
            delta)))

init
 hide({conftau,callvalue,callresult,buffercall},
  encap({handle_call,gen_server_call,
         handle_cast,gen_server_cast,
           gscall,gshcall,gscast,gshcast,
           gen_server_reply,gen_server_replied,
           rcallresult, wcallresult, rcallvalue, wcallvalue},
      cd_storage_init(cd,cons(cons(tuple(m1,
      tuplenil(cons(int(s(0)),cons(int(s(s(0))),nil)))),
       cons(tuple(m2,
      tuplenil(cons(int(s(0)),cons(int(s(s(0))),nil)))),nil)),
      cons(int(s(s(0))),cons(int(s(s(s(0)))),cons(int(s(0)),
                                                  nil)))))
      || cd_storage_buffer(cd,nomsg) ||

      tape_storage_init(tape,cons(cons(tuple(m1,
      tuplenil(cons(int(s(0)),cons(int(s(s(0))),nil)))),
         cons(tuple(m3,
      tuplenil(cons(int(s(0)),cons(int(s(s(0))),nil)))),nil)),
      cons(int(s(0)),cons(int(s(s(0))),cons(int(s(0)),nil))))
      || tape_storage_buffer(tape,nomsg) ||
      storage_group_init(storage_group,cons(tape,cons(cd,nil)))
      || storage_group_buffer(storage_group,nomsg) ||
      storage_sched_init(storage_sched,storage_group) ||
      storage_sched_buffer(storage_sched,nomsg) ||
      streaming_sched_init(streaming_sched,storage_sched) ||
      streaming_sched_buffer(streaming_sched,nomsg) ||
      users_init(pid(0),streaming_sched) ||
      callreturn(calls(cd,empty,calls(tape,empty,
          calls(storage_group,
             empty,calls(storage_sched,empty,
                  calls(streaming_sched,empty,
                 calls(pid(0),empty,nocalls)))))))
           ))
```

Appendix C

The implementation of *VoDKAV*

Contents

C.1 General design of the tool

In Section 9.4 we explained that, as a proof of concept, we developed a GUI interface with the goal of hiding the internal details of the method we proposed. In this appendix, we discuss how the tool was designed and implemented.

The implementation, completely done in *Erlang*, has a clean design based on three main layers: the first one handles a list of models, the second one lets the user change the configuration for each of the models, and the last one allows the user to check some properties.

Adding a new layer or more functionality is very simple with that structure. Also, creating a new interface would be easy. For example, in the future there are plans for developing a text based interface for the tool, and also for transforming the `gs` based interface into a more powerful one based on `GTK` using GTKNode (a project for binding `GTK` and *Erlang*).

All the information handled by the tool is stored in XML files and managed using the `xmerl` *Erlang* module for reading, modifying and writing XML files.

The process architecture of the *VoDKAV* tool can be seen in Fig. C.1. The square boxes represent *Erlang* processes that communicate passing messages. The dotted boxes represent external files read and written by the processes. The small rounded squares represent the main external tools used by *VoDKAV*.

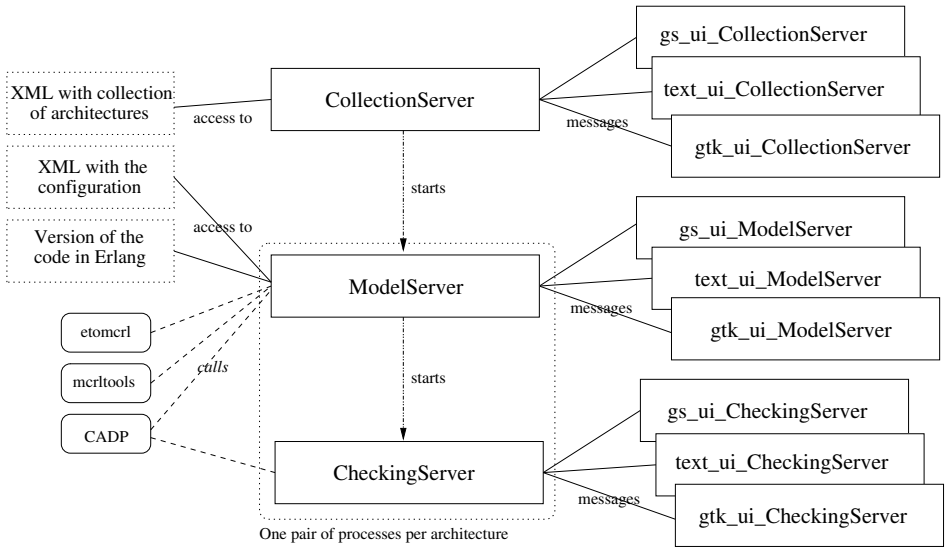


Figure C.1: Design of the *VoDKAV*: GUI interface for our method

C.2 The `CollectionServer` and its interfaces

When the application is started, the `CollectionServer` (an *Erlang* process implemented as a generic server) is created, together with one or several of the interfaces available to access it. The structure follows the Model-View-Controller design pattern.

When booting, the `CollectionServer` reads an XML file where all the available designs are declared, and stores them in an internal format as part of the server state. A design is a combination of a model version and its configuration. An example of the information contained in the XML file is shown below:

```
<?xml version="1.0" ?>

<designs>
<design>
   <name>Design number 4</name>
   <path>new-design4</path>
   <created>Sep, 2005</created>
   <modified>Jul, 2006</modified>
   <version>simple_param_bw</version>
   <description>2levels/bw</description>
</design>
<design>
...
</design>
...
</designs>
```

Name, *created*, *modified* and *description* are just meta-information about the concrete design. *path* shows to the application where to find the description of the

design with the configuration details, and *version* points to the *Erlang* source code of the concrete version of *VoDKA* this design uses.

The corresponding UI process associated to the `CollectionServer` offers the user interfaces (graphical or text based) for consulting, adding, or removing designs from the application catalog. That user actions are transformed into standard *Erlang* messages that are understood by the `CollectionServer`, which is the process actually doing all the management of the design information.

C.3 The `ModelServer` and its interfaces

From the user interface, the user can select one of the designs and start the second part of the system, the `ModelServer` with its set of associated interfaces (text or graphical ones). This process is again implemented as a generic server. When booting, receives from the `CollectionManager` the relevant information about the source code version and the place where the information about the design configuration can be found. This information is read from another XML file and stored as part of the state of the server. An example of the file structure can be seen in the following example:

```
<?xml version="1.0"?>
<design>
     <levels>
          <level>
               <name>streaming</name>
               <type>streaming_level</type>
               <max_connections>10</max_connections>
               <max_bandwidth>20</max_bandwidth>
               <cost>50</cost>
               <next>storage</next>
          </level>
          <level>
               <name>storage_group</name>
               <type>storage_group</type>
               <max_connections>10</max_connections>
               <max_bandwidth>15</max_bandwidth>
               <cost>70</cost>
               <next>devices</next>
          </level>
     </levels>
     <storages>
          <storage>
               <name>tape</name>
               <type>tape_storage</type>
               <max_connections>1</max_connections>
               <max_bandwidth>2</max_bandwidth>
               <cost>1</cost>
               <mo_list>
                    <mo>m1</mo>
                    <mo>m3</mo>
               </mo_list>
          </storage>
```

```
          <storage>
               <type>cd_storage</type>
               <max_connections>2</max_connections>
               <max_bandwidth>3</max_bandwidth>
               <cost>1</cost>
               <mo_list>
                    <mo>m1</mo>
                    <mo>m2</mo>
               </mo_list>
          </storage>
     </storages>
</design>
```

The user can start as many `ModelManagers` as needed, in order to work at the same time with different designs. If one of them crashes, this does not affect to the rest, and the process crashed can be restarted in a very clean way.

From any of the interfaces offered to the user, a lot of operations can be done over the design. The supervision tree can be drawn for the current configuration. The model can be translated to μ`CRL` and then the state graph generated. Then the state space or the architecture graph can be shown, or the current information saved at any time to the XML files. Again, when the users selects any option, it is translated into a message and sent to the `ModelManager` process associated to the model we are working with. Then the generic server calls the call-back module, where the message is parsed and the different tools called with the rigth parameters (in most of the cases generated on-demand from the state of the model).

The following code shows part of the call-back module for the `ModelManager`:

```
handle_cast (draw, {#designData{storages=StorageList,
                                levels=Levels},
                          Dir, ErlModelVersion}) ->
    ModelDir = ?VODKAVDir ++ "/" ++ ?ERLANGModelDir ++
            "/" ++ErlModelVersion,
    {ok, PreviousDir} = file:get_cwd(),
    StorageListPrepared = prepare_for_etomcrl(StorageList),
    file:set_cwd(ModelDir),
    spawn(visualize,supervisor,[vodka,start_link,
                                  [StorageListPrepared]]),
    receive after 1000 -> ok end,
    file:set_cwd(PreviousDir),
    {noreply, {#designData{storages=StorageList,
                       levels=Levels}, Dir, ErlModelVersion}};
```

It shows what happens when from the user interface we receive a message requesting the current model to be drawn. Basically, what is done is to prepare the relevant part of the state (in this case the list of storages) for passing it as parameter to the `visualize` module, and then calling it with the right arguments.

A bit more complex is the part of the call-back module that handles the `generate` message, received when the user wants to create the μ`CRL` translation and then the whole state space and the reduced one:

```
handle_cast (generate,
```

```
        {#designData{storages=StorageList, levels=Levels},
                              Dir, ErlModelVersion}) ->
    StorageListPrepared = prepare_for_etomcrl(StorageList),

    {ok, PreviousDirectory} = file:get_cwd(),
    file:set_cwd(?VODKAVDir++"/"++
           ?ERLANGModelDir++"/"++"/"++ErlModelVersion),

    PreviousMCRLDirEnv = os:getenv("MCRLDIR"),
    os:putenv("MCRLDIR",?VODKAVDir++"/"++?DESIGNSDir++"/"++Dir),

    PreviousTEMPDirEnv = os:getenv("MCRLTEMP"),
    os:putenv("MCRLTEMP",?VODKAVDir++"/"++?DESIGNSDir++"/"++Dir),

    etomcrl:supervisor(".",?VODKAVDir++"/"++?DESIGNSDir++"/"++
      Dir,vodka,start_link,
               [StorageListPrepared],[{file,"/tmp"}]),

    file:set_cwd(?VODKAVDir++"/"++?DESIGNSDir++"/"++Dir),

    io:format("~s",[os:cmd("mv vodka.mCRL vodka-det.mCRL")]),

    Pattern =
"\\(" ++ "gen_server_call(StreamingScheduler,tuple(lookup,
                        tuple(" ++ "\\)" ++
"\\(" ++ "m1" ++ "\\)" ++
"\\(" ++ ", tuplenil(" ++ "\\)" ++
"\\(" ++ ".*" ++ "\\)" ++
"\\(" ++ "))),MCRLSelf)" ++ "\\)" ++
"\\(" ++ "." ++ "\\)",
    AllMOs = sets:to_list (get_MOs_from_storages(StorageList)),
    ReplacementPattern = sed_mo_pattern(AllMOs),
    Command = "sed -e 's/"++
Pattern++ "/" ++
ReplacementPattern++ "/' " ++
"vodka-det.mCRL" ++ " > " ++
"vodka.mCRL",
    io:format("~s~n",[os:cmd(Command)]),

    io:format("~s",[os:cmd("time " ++
                      ?MCRL_Command ++ " -tbfile vodka.mCRL")]),
    io:format("~s",[os:cmd("time instantiator vodka.mCRL")]),
    transitions:file("vodka.mCRL.aut"),

    io:format("~s",[os:cmd("cp " ++ ?VODKAVDir ++
                           "/" ++ ?SCRIPTSDir ++ "/"
   ++ ?ABSTRACTScript ++ " .")]),
    io:format("~s",[os:cmd("cp "++
                           ?VODKAVDir ++ "/" ++
                           ?SCRIPTSDir ++ "/"
   ++ ?SIMPLIFYScript ++ " .")]),
    io:format("~s",[os:cmd("time svl " ++ ?ABSTRACTScript)]),
    io:format("~s",[os:cmd("rm -rf " ++ ?ABSTRACTScript
```

```
                                    ++ " " ?SIMPLIFYScript)]),

    file:set_cwd(PreviousDirectory),
    case PreviousMCRLDirEnv of
false -> ok;
_ -> os:putenv("MCRLDIR", PreviousMCRLDirEnv)
    end,
    case PreviousTEMPDirEnv of
false -> ok;
_ -> os:putenv("MCRLTEMP", PreviousTEMPDirEnv)
    end,
    {noreply, {#designData{storages=StorageList,
                           levels=Levels},
       Dir, ErlModelVersion}};
```

The main steps performed in the code above are: first, calling the `etomcrl` tool; second, using the UNIX `sed` command in order to modify the μ`CRL` specification for including in the user request all the possible combinations of *Media Objects* and bandwidths; third, calling the instantiator in order to generate the whole state space for the μ`CRL` file; and forth, using a SVL script that calls several tools of the `CADP` toolset in order to generate the reduced graph. In the middle of these four steps, different tasks related to the environment management are performed, and the measurements about the amount of time it takes to execute each of the tools is provided to the user.

C.4 The `CheckingServer` and its interfaces

When the user finished doing all kind of operations with the current model, the third part of the *VoDKAV* tool can be started: the `CheckingServer` and its associated graphical or text based user interfaces. From the interface, several standard or user provided properties can be selected for checking. This is again translated into messages that are sent to the generic server in charge of first constructing the properties from the system configuration data necessary, and then calling the `CADP` tool for doing the model checking of the property. There are also interfaces for showing the property created or the counterexample generated if the logical formula does not hold for the model.

The following code is part of the call-back module of the generic server implementing the `CheckingServer`:

```
handle_cast({checkProperty, Property}, {Dir}) ->
    check_property(Dir,Property),
    {noreply,{Dir}};
handle_cast({diagProperty, Property}, {Dir}) ->
    diag_property(Dir, Property),
    {noreply,{Dir}};
handle_cast({viewProperty, Property}, {Dir}) ->
    view_property(Dir, Property),
    {noreply,{Dir}};
handle_cast({scenviewProperty, BeforeScenario,
                               AfterScenario},
```

```
            {Dir}) ->
    scen_property(BeforeScenario, AfterScenario),
    {noreply,{Dir}};
handle_cast({scencheckProperty, BeforeScenario,
                                AfterScenario},
            {Dir}) ->
    check_property_string(Dir,
          scen_property(BeforeScenario, AfterScenario),
                        ?scenario),
    {noreply,{Dir}};
handle_cast({checkmaxPlays, MO}, {Dir}) ->
    check_max_plays(MO,Dir,1),
    {noreply,{Dir}};
handle_cast(checkmaxPlays, {Dir}) ->
    check_max_plays(".*",Dir,1),
    {noreply,{Dir}};
handle_cast({checkadhocProperty,PropertyString}, {Dir}) ->
    check_property_string(Dir,PropertyString, ?adhoc),
    {noreply,{Dir}}.

check_property_string (Dir, PropertyString, PropertyName) ->
    {ok, File} = file:open(?VODKAVDir ++ "/" ++ ?PROPSDir
                       ++ "/" ++ PropertyName ++ ?PropExtension,
   [write]),
    io:put_chars(File, PropertyString),
    check_property (Dir, PropertyName),
    {noreply,{Dir}}.

check_property (Dir, Property) ->
    {ok, PreviousDir} = file:get_cwd(),
    file:set_cwd(?VODKAVDir ++ "/" ++ ?DESIGNSDir
                            ++ "/" ++ Dir),
    Answer = os:cmd(?CADP_CounterexampleCommand ++ " " ++
        Property ++ ?CounterexampleExtension
                 ++ " -verbose " ++
           ?VODKAVDir ++ "/" ++ ?PROPSDir ++ "/" ++
                    Property ++ ?PropExtension),
    file:set_cwd(PreviousDir),
    Answer.

diag_property (Dir, Property) ->
    {ok, PreviousDir} = file:get_cwd(),
    file:set_cwd(?VODKAVDir ++ "/" ++ ?DESIGNSDir
                            ++ "/" ++ Dir),
    os:cmd(?CADP_EditCommand ++ " " ++
                    Property ++ ?CounterexampleExtension),
    file:set_cwd(PreviousDir).

scen_property (BeforeScen, AfterScen) ->
    "[" ++ scen_property_list(BeforeScen) ++ "]" ++
"<" ++ scen_property_list(AfterScen) ++ ">"
            ++ ?true.
```

```
scen_property_list([]) -> "";
scen_property_list([MO,Bw]) ->
    "'play(.*," ++ MO ++ "," ++ Bw ++ ")'";
scen_property_list([MO|[Bw|BeforeScen]]) ->
    "'play(.*," ++ MO ++ "," ++ Bw ++ ")'." ++
scen_property_list(BeforeScen).

check_max_plays(MO,Dir,Times) ->
    {ok, File} = file:open(?VODKAVDir ++ "/" ++
              ?PROPSDir ++ "/" ++ ?maxPlays ++ ?PropExtension,
   [write]),
    io:put_chars(File, exists_playsmo_property(MO,Times)),
    case extract_boolean_from_cadp_answer(
                          check_property (Dir, ?maxPlays)) of
true ->
    check_max_plays(MO,Dir,Times+1);
false ->
    Times-1
    end.
```

The result of all this modules is a very flexible, extensible and easy to maintain and use tool. It is well adapted to its goals of proving that hiding most of the underlying concepts and tools of the method proposed in this thesis is possible. The tool has been used extensively while doing the present thesis and there are current plans for extending and improving that are here left as future work.

Appendix D

Thesis metainformation

The more relevant thesis meta-information is:

- The thesis was developed during 5 years, from mid-2001 to mid-2006.
- Of those 5 years, about one was spent in Sweden (G̈teborg and Stockholm) and four in Spain (A Corunha).
- During the thesis, research visits where done to the Ericsson Computer Science Lab (2001 and 2002), to the Swedish Institute of Computer Science (2002), to the IT-University of Göteborg (2003, 2004 and 2006), and to the Universidad Politécnica de Madrid (2005 and 2006).
- The last version of the thesis was finished June 21st, 2006.

Main tools used for carrying out the research and development:

- `Debian GNU/Linux` as operating system.
- `GNOME` as desktop environment.
- *Erlang/OTP* for the development of all the systems and applications.
- `CADP` for manipulation of state spaces and model checking.
- μ`CRL toolset` for the creation of the state space from μ`CRL`.
- `etomcrl` for the generation of μ`CRL` from the *Erlang* source code.
- `McErlang` for the generation of the state space from the *Erlang* source code.
- `emacs` as IDE for the development in the different languages.
- UNIX scripting languages, commands, and `autotools` for helping with the automation of the steps.
- `CVS` for version management.
- *VoDKA* as main motivation and case study for the research in the thesis.
- `XML` languages and libraries for data storage.

Main tools used for writing the thesis manuscript:

- `Debian GNU/Linux` as operating system.
- `GNOME` as desktop environment.
- `LaTeX` as markup language for the thesis manuscript.
- `pdflatex` for the generation of the PDF version.
- `Bibtex` and `JavRef` for bibliography management.
- `emacs` as text editor for LaTeX.
- `xfig` and `dia` for the diagrams and graphs.
- `CVS` for version management.

Appendix E

Licensing of the thesis

Bibliography

[AA98] Thomas Arts and Joe Armstrong. A practical type system for Erlang. In *Proc. of the Erlang User Conference*, Stockholm, Sweden, September 1998.

[ABC+94] Adnan Aziz, Felice Balarin, Szu-Tsung Cheng, Ramin Hojati, Timothy Kam, Sriram C. Krishnan, Rajeev K. Ranjan, Thomas R. Shiple, Vigyan Singhal, Serdar Tasiran, Huey-Yih Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. HSIS: A BDD-based environment for formal verification. In *Design Automation Conference*, pages 454–459, 1994.

[ABD04] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. on Software Tools for Technology Transfer*, 2004.

[ACD+03] Thomas Arts, Gennady Chugunov, Mads Dam, Lars-Åke Fredlund, Dilian Gurov, and Thomas Noll. A tool for verifying software written in erlang. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4), 2003.

[AD99] Thomas Arts and Mads Dam. Verifying a distributed database lookup manager written in erlang. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 682–700, London, UK, 1999. Springer-Verlag.

[AGF+03] Carlos Abalde, Víctor M. Gulías, José L. Freire, Juan J. Sánchez, and José M. García-Tizón. Development of a scalable, fault tolerant and low cost cluster-based e-payment system with a distributed functional kernel. In *Proceedings of Ninth International Conference on Computer Aided Systems Theory. LNCS 2809, selected papers*, pages 220–230, February 2003.

[AGFS02] Carlos Abalde, Víctor M. Gulías, José L. Freire, and Juan J. Sánchez. A cluster-based payment gateway system developed using a distributed functional language. In *Proceedings of EurAsia ICT 2002, Advances in Information and Communication Technology*, pages 79–83, October 2002.

[AH03] T. Arts and J. Hughes. Erlang QuickCheck. In Ericsson, editor, *Proceedings of the Ninth International Erlang/OTP User Conference (EUC 2003)*, Stockholm, Sweden, November 2003.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[AK04] Reynald Affeldt and Naoki Kobayashi. A Coq library for verification of concurrent programs. *Electronic Notes in Theoretical Computer Science*, 2004.

[AKY05] Reynald Affeldt, Naoki Kobayashi, and Akinori Yonezawa. Verification of concurrent programs using the Coq proof assistant: A case study. *IPSJ Transactions on Programming*, 46(1):110–120, 2005.

[AO92] D. Anderson and Y. Osawa. A file system for continuous media. *ACM Transactions on Computer System*, 10(4), November 1992.

[App] Apple Computer Inc. About Darwin streaming server.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[AS02] Thomas Arts and Juan J. Sánchez. Global scheduler properties derived from local restrictions. In *Proceedings of the ACM Sigplan Erlang Workshop at the Principles,Logics, and Implementations of high-level programming languages*, Pittsburg, USA, October 2002. ACM.

[AVWW96] J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming in Erlang, 2nd edition*. Prentice Hall International, 1996.

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BC87] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. In *OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming*, September 1987.

[Ber94] S. Berson. Staggered striping in multimedia information systems. *ACM SIGMOD Conference*, pages 79–90, June 1994.

[BFD05] C. Benac Earle, L. Fredlund, and J. Derrick. Verifying fault-tolerant Erlang programs. In *ACM SIGPLAN Erlang workshop*, pages 26–34, 2005.

[BFGS01] Miguel Barreiro, José Luis Freire, Víctor M. Gulías, and Juan J. Sánchez. Exploiting sequential libraries on a cluster of computers. In *Proceedings of the Erlang Workshop, (EW 2001)*, Firenze, Italy, September 2001.

[BG99] M. Barreiro and V. M. Gulías. Cluster setup and its administration. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume I. Prentice Hall, 1999.

[BGF+01] Miguel Barreiro, Víctor M. Gulías, José L. Freire, Javier Mosquera, and Juan J. Sánchez. An Erlang-based hierarchical distributed VoD system. In *Proc. of Seventh International Erlang/OTP User Conference*, Stockholm, Sweden, September 2001. Ericsson Utvecklings AB.

[BGM95] S. Berson, L. Golubchik, and R.R. Muntz. A fault tolerant design of a multimedia server. *ACM SIGMOD Conference*, 1995.

[BGMS01] Miguel Barreiro, Víctor M. Gulías, Javier Mosquera, and Juan J. Sánchez. Utilización de programación funcional distribuida y clusters linux en el desarrollo de servidores de vídeo bajo demanda. In Senén Barro Ameneiro, José Luis Freire Nistal, and Jesús Rivero Laguna, editors, *Actas del Simposio en Informática y Telecomunicación, SIT 2001*, pages 83–95, September 2001.

[BGS00] M. Barreiro, V. Gulías, and J.J. Sánchez. A monitoring and instrumentation tool developed in Erlang. In *Proc. of 6th International Erlang/OTP User Conference*, Stockholm, Sweden, 2000.

[BGSJ01] Miguel Barreiro, Victor M. Gulías, Juan J. Sánchez, and J. Santiago Jorge. The tertiary level in a functional cluster-based hierarchical VoD server. In *Proc. of EUROCAST*, pages 540–554, 2001.

[BH06] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods ...ten years later. *Computer*, 39(1):40–48, 2006.

[BM88] Robert S. Boyer and J. Strother Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[BMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, 2004.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, New York, 1996.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[BR98a] D. Baumer and D. Riehle. Product trader. In *In Pattern Languages of Program Design 3 (PLoPD3)*, pages 29–46. Addison-Wesley, 1998.

[BR98b] S. Blau and J. Rooth. AXD 301 – a new generation ATM switching system. *Ericsson Review*, 1, 1998.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BS95] Ulrich Berger and Helmut Schwichtenberg. Program extraction from classical proofs. In *LCC '94: Selected Papers from the International Workshop on Logical and Computational Complexity*, pages 77–97, London, UK, 1995. Springer-Verlag.

[BT85] Jan A. Bergstra and J. V. Tucker. Top-down design and the algebra of communicating processes. *Sci. Comput. Program.*, 5(2):171–199, 1985.

[CCF+] Cristina Cornes, Judical Courant, Jean-Christophe Fillitre, Gérard Huet, Pascal Manoury, César Mu noz, Chetan Murthy, Catherine Parent, and et al. The Coq proof assistant - reference manual.

[CDH00] J. Corbett, M. Dwyer, and L. Hatcliff. Bandera: A source-level interface for model checking java programs. In *Teaching and Research Demos at ICSE'00*, June 2000.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGR93] D. Craigen, S. Gerhart, and T.J. Ralston. An international survey of industrial applications of formal methods (volume 1: Purpose, approach, analysis and conclusions, volume 2: Case studies). Technical Report NIST GCR 93/626-V1 & NIST GCR 93-626-V2, National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA, 1993.

[CGR95] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, 1995.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings*

of the fifth ACM SIGPLAN international conference on Functional programming, pages 268–279, New York, NY, USA, 2000. ACM Press.

[CKK02] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures. Methods and Case Studies*. Addison-Wesley, 2002.

[CKM+91] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. Eves: An overview. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 389–405, London, UK, 1991. Springer-Verlag.

[CKY94] M. Chen, D. Kandlur, and P. S. Yu. Support for fully interactive playout in a disk-array-based video server. *Proceedings of the 2nd Annual ACM Multimedia Conference*, October 1994.

[Col89] M. Cole. Algorithmic skeletons: Structured management of parallel computation. *Research Monographs in Parallel and Distributed Computing*, 1989.

[CPS93a] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CPS93b] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[CS00] Inc. Cisco Systems. *A Distributed Video Server Architecture for Flexible Enterprise-Wide Video Delivery. White Paper*. 2000.

[CS05] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proc. of ACM SIGPLAN Erlang Workshop*, 2005.

[CT97a] S. G. Chan and F. Tobagi. Hierarchical storage systems for interactive Video-on-Demand. *Technical Report, Stanford University, Computer Systems Laboratory, Number CSL-TR-97-723*, 1997.

[CT97b] S. Gary Chan and F. Tobagi. Hierarchical storage systems for interactive Video-on-Demand. *Technical Report, Stanford University, Computer Systems Laboratory, Number CSL-TR-97-723, p. 84.*, April 1997.

[CVV97a] T. Chiueh, C. Venkatramani, and M. Vernick. Design and implementation of the Stony Brook Video Server. *Software – Practice and Experience*, 1997.

[CVV97b] T. Chiueh, M. Vernick, and C. Venkatramani. Performance evaluation of Stony Brook Video Server. *ECSL-TR-24*, 1997.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[CZ92] Edmund Clarke and Xudong Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. Technical Report CS-92-147, 1992.

[D00] Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. Licentiate Thesis TRITA-IT AVH 00:08, Royal Institute of Technology, Stockholm, Sweden, November 2000.

[DF98] Mads Dam and Lars-Åke Fredlund. On the verification of open distributed systems. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 532–540, New York, NY, USA, 1998. ACM Press.

[DFG98] Mads Dam, Lars-Åke Fredlund, and Dilian Gurov. Toward parametric verification of open distributed systems. *Lecture Notes in Computer Science*, 1536:150–185, 1998.

[DHLV96] D. Du, J. Hsieh, J. Liu, and R. J. Vetter. *Building Video-on-Demand Servers Using Shared-Memory Multiprocessor.* North Dakota State University, 1996.

[DHT04] Tai Do, Kien A. Hua, and Mounir Tantaoui. P2VoD: Providing fault tolerant Video-on-Demand streaming in Peer-to-Peer environment. In *Proc. of the IEEE Int. Conf. on Communications*, June 2004.

[dMvEP01] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers - SPARKLE: A functional theorem prover. In Arts and Mohnen, editors, *Proc. of the 13th International Workshop on the Implementation of Functional Languages (IFL 2001)*, pages 55–71, älvsjö, Sweden, 2001. Springer-Verlag, LNCS 2312.

[DSB04] Jim Davies, Wolfram Schulte, and Michael Barnett, editors. *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, volume 3308 of *Lecture Notes in Computer Science*. Springer, 2004.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL - formal object-oriented language for communicating systems.* Prentice Hall, 1997.

[Eks00] U. Ekstrom. Design patterns for simulation in Erlang/OTP. Master's thesis, University of Upsala, Upsala, Sweden, 2000.

[Eri06] Ericsson. Erlang/OTP R11B documentation, 2006.

[Erl00] L. Erlikh. Leveraging legacy system dollars for E-business. *IT Pro*, pages 17–23, May/June 2000.

[FBA03] Roy Friedman, Lior Baram, and Shiri Abarbanel. Fault-tolerant multi-server Video-on-Demand service. *International Parallel and Distributed Processing Symposium (IPDPS'03)*, 00:70a, 2003.

[FBE06] Lars-Åke Fredlund and Clara Benac-Earle. Model checking erlang programs: The functional approach. In *ACM Sigplan International Erlang Workshop*, Portland, Oregon, USA, September 2006.

[FG99] Lars-Åke Fredlund and Dilian Gurov. A framework for formal reasoning about open distributed systems. In *Asian Computing Science Conference*, pages 87–100, 1999.

[FGKM96] J.C. Fernández, H. Garavel, A. Kerbrat, and R. Mateesc. Caesar/Aldébaran development package: A protocol validation and verification toolbox. In *11th Int. Conf. on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, August 1996.

[FGN+03] L-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *Int. J. on Software Tools for Technology Transfer*, 4(4):405–420, 2003.

[Fis96] Kathryn Fisler. *A unified approach to hardware verification through a heterogeneous logic of design diagrams*. PhD thesis, Indiana University, 1996. Co-Chairman-K. Jon Barwise and Co-Chairman-Steven D. Johnson.

[FNBFFBS01] J. L. Freire-Nistal, A. Blanco-Ferro, J.E. Freire-Brañas, and Juan J. Sánchez. Fusion and deforestation in Coq. In *Eurocast 2001, selected papers. Lecture Notes in Computer Science 2178*, pages 583–596, Las Palmas de Gran Canaria, España, February 2001. Springer-Verlang.

[Fok00] Wan Fokkink. *Introduction to Process Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[Fre01] L-Å. Fredlund. A framework for reasoning about Erlang code. PhD thesis, Dept. of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, 2001.

[GAS03a] Victor M. Gulías, Carlos Abalde, and Juan J. Sánchez. Lambda goes to hollywood. In *Practical Aspects of Declarative Languages (PADL 2003). Lecture Notes of Computer Science 2562*, January 2003.

[GAS03b] Víctor M. Gulías, Carlos Abalde, and Juan J. Sánchez. Lambda goes to Hollywood. In *Fifth International Simposium on Practical Aspects of Declarative Languages (PADL'03)*, volume 2562 of *LNCS*. Springer-Verlang, January 2003.

[GBF05] V. Gulías, M. Barreiro, and J. L. Freire. VoDKA: Developing a Video-on-Demand server using distributed functional programming. *Journal of Functional Programming*, 15 (3):403–430, 2005.

[GC92] Gemmel and Christodoulakis. Principles of delay-sensitive multimedia data storage and retrieval. *ACM Transaction on Information Systems*, 10(1), January 1992.

[GCSPMR04] Alejandro García-Castro, Juan José Sánchez-Penas, and Fco. Javier Morán-Rua. SERVAL: a VLAN software switch developed in Erlang. In *Proceedings of Erlang/OTP User Conference*, Stockholm, Sweden, October 2004. Ericsson AB.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1994.

[GL01a] Hubert Garavel and Frédéric Lang. SVL: a scripting language for compositional verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[GL01b] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Technical Report SEN-R0117, CWI, Amsterdam, The Netherlands, 2001.

[GLM02] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GR01] J. F. Groote and M. A. Reniers. Algebraic process verification. In *Handbook of Process Algebra*, pages 1151–1208. Elsevier, 2001.

[Gro93] The RAISE Language Group. *The RAISE specification language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Gro97] J.F. Groote. The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, Amsterdam, The Netherlands, June 1997.

[GTBRS01] José M. García-Tizón, Antonio Blanco, Miguel Rodríguez, and Juan J. Sánchez. Integración y coexistencia de aplicaciones legacy y aplicaciones web usando patrones. In *Actas del Simposio en Informática y Telecomunicación (SIT 2001)*, pages 109–121, Septiembre 2001.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HD01] John Hatcliff and Matthew Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. *Lecture Notes in Computer Science*, 2154, 2001.

[HF04] Yin-Fu Huang and Chih-Chiang Fang. Load balancing for clusters of VoD servers. *Inf. Sci. Inf. Comput. Sci.*, 164(1-4):113–138, 2004.

[HLS+02] Pao-Ann Hsiung, Trong-Yen Lee, Win-Bin See, Jih-Ming Fu, and Sao-Jie Chen. VERTAF: An object-oriented application framework for embedded real-time systems. In *Proc. of the International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2002)*. IEEE Computer Science Press, April 2002.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Hol91] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., 1991.

[HP00] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, March 2000.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[HS96] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer-Verlag, 1996.

[HS04] F. Huch and V. Stolz. Runtime verification of Concurrent Haskell programs. In *Proceedings of the Fourth Workshop on Runtime Verification*, to appear in ENTCS. Elsevier Science Publishers, 2004.

[Huc99] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, September 1999.

[Huc01] F. Huch. Model checking erlang programs - abstracting recursive function calls. In *Proc. of the International Workshop on Functional and Constraint Logic Programming (WFLP 2001)*, Kiel, Germany, 2001.

[ISO88] ISO/IEC. LOTOS – A formal description technique based on the temporal ordering of observational behaviour. In *International Standard 8807*, Information Processing Systems – Open Systems Interconnection. International Organization for Standardization, September 1988.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[Kie02] Richard B. Kieburtz. P-logic: property verification for Haskell programs, 2002.

[KM96] M. Kaufmann and J. Moore. ACL2: An industrial strength version of Nqthm. In *Proc. of the Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Computer Society Press, June 1996.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[Kur98] R.P. Kurshan. *FormalCheck User's Manual*. Cadence Design, Inc., 1998.

[KZ95] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[LAKN98] Shaoying Liu, Masashi Asuka, Kiyotoshi Komaya, and Yasuaki Nakamura. An approach to specifying and verifying safety-critical systems with practical formal method SOFL. In *Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pages 100–114, Monterey, California, USA, August 1998. IEEE Computer Society Press.

[LC03] Yiu-Wing Leung and T.K.C. Chan. Design of an interactive Video-On-Demand system. *IEEE Transactions on Multimedia*, 5:130–140, March 2003.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[LL96] S. Lau and J. Lui. Scheduling and replacement policies for a hierarchical multimedia storage server. *Proceedings of Multimedia Japan 96, International Symposium on Multimedia Systems*, March 1996.

[LL97] U. Lechner and C. Lengauer. Specification, abstraction and verification in a concurrent object-oriented language. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming*, pages 544–547, 1997.

[LP99] Lamport and Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21, 1999.

[LS93] P. Lougher and D. Shepherd. The design of a storage server for continous media. *The Computer Journal*, 36(1), 1993.

[LS04] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, November 2004.

[LV94] Thomas D. C. Little and Dinesh Venkatesh. Prospects for interactive Video-on-Demand. *IEEE MultiMedia*, 1(3):14–24, Fall 1994.

[MBB+99] Zohar Manna, Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Mark Pichora, Henny B. Sipma, and Tomás E. Uribe. An update on STeP: Deductive-algorithmic verification of reactive systems. In Rudolf Berghammer and Yassine Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 174–188. Springer-Verlag, 1999.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[MGP+01] Castor Mariño Pérez, Víctor M. Gulías, Marta Penas, M. G. Penedo, V. Leborán, A. Mosquera, M. J. Carreira, and D. Lloret. Sistema de interpretación automática de secuencias slo basada en un servidor vod. *Simposio en Informática y Telecomunicación*, pages 319–329, September 2001.

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil95] Robin Milner. *Communication and concurrency.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MV96] T. Chiueh M. Vernick, C. Venkatramani. Adventures in building the Stony Brook Video Server. *Proceedings of ACM Multimedia*, 1996.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Ora98a] Oracle. *Oracle Video Server Administrators Guide and Command Reference*. 1998.

[Ora98b] Oracle. *Oracle Video Server System Technical Overview*. 1998.

[OSR95] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System*. CSL, 1995.

[PA03] Juan José Sánchez Penas and Thomas Arts. Performance analysis using model checking. In *Proceedings of the 9th Erlang User Conference (EUC 2003)*. Ericsson, November 2003.

[PAI02] L. Pinho, C. Amorim, and E. Ishikawa. GloVE: A distributed environment for low cost scalable VoD systems. *14th Symposium on Computer Architecture and High Performance Computing*, 2002.

[Phi] Philips. Webcine server guide.

[Pie97] Benjamin Pierce. Foundational calculi for programming languages. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Univ. of Edinburgh, 1994.

[PR03] Juan José Sánchez Penas and Carlos Abalde Ramiro. Extending the VoDKA architecture to improve resource modelling. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 15–22, New York, NY, USA, 2003. ACM Press.

[Pre97] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1997.

[QS01] Juan Quintela and Juan J. Sánchez. Persistent Haskell. In *Eurocast 2001, selected papers. Lecture Notes in Computer Science 2178*, pages 657–667. Springer-Verlang, February 2001.

[RdS91] V. Roy and Robert de Simone. Auto/autograph. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 65–75, London, UK, 1991. Springer-Verlag.

[RGA+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. -T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[Rus93] J. Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-937, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[RV92] P. Rangan and H. Vin. Designing an on-demand multimedia service. *IEEE Communication Magazine*, 30(7), July 1992.

[RV93] P. Rangan and H. Vin. Efficient storage techniques for digital continuous multimedia. *Transactions on Knowledge and Data Engineering*, August 1993.

[SÓ1] J.J. Sánchez. State of the art in formal verification of distributed functional software. In *Report for the Diploma de Estudios Avanzados*. Universidade da Corunha, 2001.

[SBG+01] Juan J. Sánchez, Miguel Barreiro, Víctor M. Gulías, José L. Freire, and Javier Mosquera. Functional scheduling in a distributed VoD server. In Thomas Arts and Markus Mohnen, editors, *Proceedings of 13th International Workshop on the Implementation of Functional Languages*, September 2001.

[Sca98] B. Scattergood. *FDR: User Manual and Tutorial*. Formal Systems (Europe) Ltd., 1998.

[SEN99] SEN group. A language and tool set to study communicating processes with data. Technical report, CWI, February 1999.

[SFB+00] Juan J. Sánchez, Jose L. Freire, Miguel Barreiro, Víctor M. Gulías, and Javier Mosquera. An Erlang-based hierarchical distributed VoD system. In *In proceedings of 7th International Erlang/OTP User Conference*, 2000.

[SGVM00] J.J. Sánchez, V. Gulías, A. Valderruten, and J. Mosquera. State of the art and design of VoD systems. In *Proc. of the International Conference on Information Systems Analysis (SCI'00-ISAS'00)*, Orlando, Florida, USA, July 2000.

[SJGFS05] J. Santiago-Jorge, V. Gulías, J.L. Freire, and J.J. Sánchez. Towards a certified and efficient computing of gröbner bases. In *Computer Aided Systems Theory - EUROCAST 2005: 10th International Conference on Computer Aided Systems Theory. LNCS 3643, revised selected papers*, pages 111–120. Springer-Verlang, February 2005.

[SJO+05] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Thomas F. Melham, Mark Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.

[SM97] Ph. Wadler S. Marlow. A practical subtyping system for erlang. In *Proc. of the Int. Conf. on Functional Programming*, Amsterdan, The Netherlands, 1997.

[SMC+95] B. Steffen, T. Margaria, A. ClaBetaen, V. Braun, and M. Reitenspiess. An environment for the creation of intelligent network services. In *Proc. of the IEC Annual Review of Communications - also SNI-Rep (Invited Contribution)*, 1995.

[SPA03] Juan José Sánchez-Penas and Thomas Arts. VoDkaV tool: Model checking or extracting global scheduler properties from local restrictions. In *Poster and Tool Presentation. Proceedings of Third International Conference on Application of Concurrency to System Design (ACSD 2003)*, pages 247–248. IEEE, June 2003.

[STH06] Simon Sheu, Wallapak Tavanapong, and Kien A. Hua. A scalable cost-effective video broadcasting system for on-demand video services. *Multimedia Tools Appl.*, 28(3):321–345, 2006.

[Sun] Sun Microsystems Inc. Sun storedge media central streaming server.

[TA04] Juan José Sánchez Thomas Arts, Clara Benac. From Erlang to μcrl. making industrial code available for research tools. In IEEE, editor, *Proceedings of Forth International Conference on Application of Concurrency to System Design (ACSD 2004)*, pages 135–144, June 2004.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[THS03] Duc A. Tran, Kien A. Hua, and Simon Sheu. A new caching architecture for efficient Video-on-Demand services on the Internet. In *SAINT '03: Proceedings of the 2003 Symposium on Applications and the Internet*, page 172, Washington, DC, USA, 2003. IEEE Computer Society.

[VGM+00] A. Valderruten, V.M. Gulías, J. Mosquera, J.J. Sánchez, and A. Blanco. Evaluación del rendimiento de un sistema de tiempo

real multihilo usando un modelo reactivo síncrono. In *Actas del Simposio Español de Informática Distribuída (SEID 2000)*, pages 437–444, Ourense, Spain, September 2000.

[VGS+01a] Alberto Valderruten, Víctor M. Gulías, Juan J. Sánchez, José L. Freire, and Javier Mosquera. Implementación de un modelo de monitorización para un servidor de vídeo bajo demanda en Erlang. In Jonás Montilva, editor, *Proceedings of XXVII Conferencia Latinoamericana de Informática*, September 2001.

[VGS+01b] Alberto Valderruten, Víctor M. Gulías, Juan J. Sánchez, José L. Freire, and Javier Mosquera. Implementación de un modelo de monitorización para un servidor de vídeo bajo demanda en Erlang. In *Proceedings of XXVII Conferencia Latinoamericana de Informática*. Jonás Montilva, September 2001.

[Wad87] P. Wadler. Efficient compilation of pattern matching. In Simon Peyton-Jones, editor, *The implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.

[WD96] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[WDRW99] P. Wilkinson, M. DeSisto, H. Rother, and Y. Wong. *IBM VideoCharger 101. IBM Redbook*. International Technical Support Organization, 1999.

[Wou01] A. G. Wouters. Manual for the μCRL tool set. Technical Report SEN-R0130, CWI, Amsterdam, The Netherlands, 2001.

[WS98] Lloyd G. Williams and Connie U. Smith. Performance evaluation of software architectures. In *WOSP '98: Proceedings of the 1st international workshop on Software and performance*, pages 164–177, New York, NY, USA, 1998. ACM Press.

[Yu98] S. Yu. *Formal Verification of Concurrent Programs Based on Type Theory*. PhD thesis, Department of Computer Science, University of Durham, October 1998.