Integrating formal reasoning into component-based approach to reconfigurable distributed systems.

Alessandro Basso

School of Social Electronics and Computer Science

INTEGRATING FORMAL REASONING INTO A COMPONENT-BASED APPROACH TO RECONFIGURABLE DISTRIBUTED SYSTEMS

By

Alessandro Basso

A thesis submitted in partial fulfilment of the requirements of the University of Westminster for the degree of Doctor of Philosophy

July 2010

Contents

List of Tables

List of Figures

Abstract

Distributed computing is becoming ubiquitous in recent years in many areas, especially the scientific and industrial ones, where the processing power - even that of supercomputers - never seems to be enough. Grid systems were born out of necessity, and had to grow quickly to meet requirements which evolved over time, becoming today's complex systems. Even the simplest distributed system nowadays is expected to have some basic functionalities, such as resources and execution management, security and optimization features, data control, etc. The complexity of Grid applications is also accentuated by their distributed nature, making them some of the most elaborate systems to date. It is often too easy that these intricate systems happen to fall in some kind of failure, it being a software bug, or plain simple human error; and if such a failure occurs, it is not always the case that the system can recover from it, possibly meaning hours of wasted computational power.

In this thesis, some of the problems which are at the core of the development and maintenance of Grid software applications are addressed by introducing novel and solid approaches to their solution. The difficulty of Grid systems to deal with unforeseen and unexpected circumstances resulting from dynamic reconfiguration can be identified. Such problems are often related to the fact that Grid applications are large, distributed and prone to resource failures. This research has produced a methodology for the solution of this problem by analysing the structure of distributed systems and their reliance on the environment which they sit upon, often overlooked when dealing with these types of scenarios. It is concluded that the way that Grid

applications interact with the infrastructure is not sufficiently addressed and a novel approach is developed in which formal verification methods are integrated with distributed applications development and deployment in a way that includes the environment. This approach allows for reconfiguration scenarios in distributed applications to proceed in a safe and controlled way, as demonstrated by the development of a prototype application.

Declaration

The work included in this thesis is the author's own. No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

External publications directly related to this thesis

In Books:

[1] A. Basso, A. Bolotov, and V. Getov, Behavioural Model of Component-based Grid Environments. In From Grids To Service and Pervasive Computing. Edited by T. Priol and M. Vanneschi, Springer, 2008.

[2] A. Basso and A. Bolotov, Towards GCM Re-Configuration - Extending Specification by Norms. In Making Grids Work, Edited by M. Danelutto, P. Fragopoulou and V. Getov, Springer, 2007.

In Proceedings:

[3] A. Bolotov, A. Basso and O. Grigoriev, Deontic extension of deductive verification of component model: Combining computation tree logic and deontic logic in natural deduction style calculus, in International Indian Conference on Artificial Intelligence, 2009.

[4] A. Basso, A. Bolotov, and V. Getov, Temporal specification and deductive verification of a distributed component model and its environment, in Secure System Integration and Reliability Improvement, (Shanghai, China), pp. 379-386, IEEE Computer Society, 2009.

[5] A. Basso, A. Bolotov, and V. Getov, State-based behavior specification for gcm systems, in The 16th Workshop on Automated Reasoning, (ARW 2009), 2009.

[6] A. Basso, A. Bolotov, and V. Getov, Automata-based formal specification of stateful systems, in The 15th Workshop on Automated Reasoning, (ARW 2008), 2008.

[7] A. Basso and A. Bolotov, Deductive verification of gcm: Deontic temporal resolution, in CoreGRID WP3 Programming model Institute, plenary meeting, 2008.

[8] A. Basso and A. Bolotov, Verification tool - towards an analysis of complexity, in The 14th Workshop on Automated Reasoning, (ARW 2007), 2007.

[9] A. Basso, A. Bolotov, and M. Urbanski, Specification and verification of reconfiguration protocols in grid component systems, in 3rd IEEE Conference On Intelligent Systems, IS 2006.

[10] A. Basso and A. Bolotov, Specification and verification of reconfiguration protocols in grid component systems, in The 13th Workshop on Automated Reasoning, (ARW 2006), 2006.

In Technical Reports:

[11] A. Basso, A. Bolotov, V. Getov, and L. Henrio, Dynamic reconfiguration of gcm components, Tech. Rep. TR-0173, Institute on Programming Model, CoreGRID - Network of Excellence, 2008.

[12] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, G. Zoppi, A. Basso, A. Bolotov, F. Baude, H. Bouziane, D. Caromel, L. Henrio, C. Perez, J. Cunha, C. Michael, P. Classen, C. Lengauer, J. Cohen, S. Mc Gough, N. Currle-Linde, P. Dazzi, N. Tonellotto, J. Dunnwebber, S. Gorlatch, P. Kilpatrick, N. Ranaldo, and E. Zimeo, Proceedings of the programming model institute technical meeting 2008, Tech. Rep. TR-0138, Institute of Programming Model, CoreGRID - Network of Excellence, 2008.

[13] A. Basso, A. Bolotov, A. Basukoski, V. Getov, L. Henrio, and M. Urbanski, Specification and verification of reconfiguration protocols in grid component systems, Tech. Rep. TR-0042, Institute on Programming Model, CoreGRID - Network of Excellence, 2006.

Acknowledgements

I would like to thank my supervisors, Alexander Bolotov, Vladimir Getov and Ludovic Henrio for their invaluable help and support throughout my PhD; the CoreGrid and GridComp projects which partially funded my research, as well as the people involved with the projects with whom I have collaborated over the past years. For their collaboration in constructing my prototype I would like to thank Stavros Isaiadis and Lan Zhang. I would also like to thank Artie Basukoski for his help proofreading the thesis and his suggestions for improvement. Finally I would like to thank my family and especially my wife for being so patient while I completed my thesis.

Chapter 1

Introduction

1.1 Overview and orientation

'Grid' or 'distributed' computing, is a term that refers to a particular type of parallel computing in which multiple computers, called 'resources', are networked together in one big virtual computer, in order to harness the power of multiple machines to process large amounts of data, to perform computational intensive calculations. The term was first coined in [KF98], and has been made popular by projects such as SETI@home [ACK+02]. Since then, many projects have developed trying to construct various types of grid infrastructures, middleware, applications etc. [OMG06, CCH+08, BFH03, oE], aimed at simplifying the way grid applications are created and used, and ultimately leading to the widespread use of Grids, making the whole framework become invisible [GBT+07]. As the use of Grids is becoming more and more commonplace, the necessity to simplify usability, increase performance, reduce failure, and so forth, are becoming more and more an essential part of Grid development. From scientific applications, to image rendering, to data processing, the computing power required has increased over the years, and it appears that Moore's Law can be applied to more and more aspects of modern computing. Furthermore, if the range of applications for distributed computing continues to grow and change, the way computation is performed has to evolve as well. Users want to be

able to create an experiment during the day, leave it to run at night, and expect to find results in the morning; unfortunately, this is not always the case. In order to ease the life of the user, developers turn to new and diverse methods to cope with unforeseen problems, trying to create software which adapts to situations. 'Adaptive software' [Hig00] is a term that refers to these kind of developments, and can be found not only in the parallel computing world, but also in everyday's home computers, where software is able to detect a variety of information about the system it is running on, and choose the appropriate configuration without any user interaction. While many of these solutions are usually hard coded in each and every piece of software, and can cope with a wide range of different problems thrown at them, they are not infallible. The causes of failures in distributed systems can range from hardware defects, to operating system malfunctions; and in Grid systems in particular, because of their distributed characteristic, failures are often caused by problems with resources. In this thesis, these causes have been investigated and an approach to a solution has been created which can handle aspects not considered before, and can be adapted to a variety of distributed computing environments through the integration of formal reasoning in the Grid applications development.

Grid Enviroment

The first challenge identified in this thesis is the almost absence of research on the 'environment' where Grid application lie. It is easy to understand that distributed applications have a close relationship with the resources utilized, but there is no clear structure in their interoperability, ultimately leading to applications which could become unstable. In this research it will be analysed in detail the effect that the infrastructure has on distributed application, aiming at providing an insight on its workings and on how to approach the environment when constructing a methodology to address related problems.

Component Model

The component models utilized in the construction of Grid applications offer an understanding of the relationship between its fundamental parts as well as their relations with the environment. In this thesis component models have been analysed in order to understand what parts can

be relevant when dealing with failures in dynamically reconfiguring distributed systems. The challenge here is to decompose these models into their parts and retrieve only the useful aspects while disregarding the rest. This process is more complex than it might appear at first as some parts which need to be clear in order to be able to apply specific techniques, might be hidden or implied. The objective is to expose those aspects while ensuring that relevant others are not disregarded.

Formal Specification and Verification

In choosing a way to solve the problem of failures in Grid systems, it is difficult to identify a procedure which is able to consider parts such as a dynamic composition, or possible infrastructure restrictions. Although formal specification and verification have a record of being successfully used in software validation, it has never been used in respect to these aspects. In this thesis it has been detailed how it is possible to adapt, improve and enable theoretical procedures on formal specification and verification in large scale software systems, through the use of a specification language created for this purpose, and a verification technique which is ideal for these types of specifications.

Integration

A challenge identified in this thesis is the integration of a formal verification methods with software systems development and deployment. As this process has to be reliable and ultimately automated in a tool, as well as being adaptable to the peculiar structure of component-based distributed systems which rely on a range of resources, a technique for integration needs to be developed with rigid constraints. The objective in this thesis will be to provide such an innovative procedure based on solid and well studied techniques.

Reconfiguration

The reconfiguration of component-based Grid applications, has been identified in this thesis as a flawed procedure in standard developments due to its failure prone aspects in composition and deployment. The construction of these distributed systems, comprised of hierarchical components, distributed deployments, resource dependent and so forth, will be affected by unforeseen

problems during dynamic reconfigurations (as in the case of suddenly unavailable resources or services), which are just not feasible to be solved only through standard software safety protocols. It is the aim of this thesis to outline a complete methodology which bypasses the software code and looks deeper into the interaction between elements of the overall system, and provides a solution which is generic enough that can be adapted to a wide range of developments in the field.

Overall Aim

The overall aim of this research is to provide, both in a theoretical way as well as by proof of concept, a verification engine capable to work simultaneously, integrating and interacting with a distributed system; capable of providing both developers and users with a way to verify dynamic reconfigurations of a grid system. This would include the distributed application, as well as all the resources which are part of the overall grid environment - such as nodes, databases, services and any other parts which are needed by the application in order to perform correctly. On a functional level, this research makes use of a combination of a newly constructed specification language as well as a suitable verification method to be the base for a tool capable to interact with the distributed system by receiving information from it, as well as applying changes and adapting to changes made by the system or the tool itself. When a user might fire a reconfiguration procedure, the tool will then be able to consider these changes and analyse whether they are appropriate for the system. Furthermore, the tool needs to adapt to the newly created configuration, and be able to take them into consideration if the need for another reconfiguration arises in the future. As an example consider a user adding the functionality for a grid to utilise wireless devices as resources. If the tool confirms that such reconfiguration is not in conflict with the overall grid environment (which, for example, might be missing a wireless adapter), the tool should then adapt to any services these new devices may now offer and may be exposed to the grid in a consequent reconfiguration.

1.2 Thesis Organization

Throughout this thesis, the related work is often interwoven with the author's contribution, due to the fact that this thesis has a broad coverage of topic areas. To ease the reader's understanding of which parts in the thesis are related material and which are the author's contribution, the contributions are highlighted in the following thesis organisation, as well as at the beginning of relevant sections throughout the body of the thesis. In chapter §2 the approach to the problem of integration of formal tools with Grid applications development and usage is outlined, and the author's approach to the solution is given. In section §2.1 model abstraction is defined, given an overview of the approaches of abstraction for formal specification, and how this can be achieved in the context of component models and the environment where grid applications lie; while in section §2.2 the author's approach to integration of the abstract model is described. Chapter §3 begins with related works, describing the types of Grid systems and their structures in section §3.1. It is then described the use of component model developments in section §3.2 where close attention is paid to how this is achieved in the component model considered and the importance of the behaviour of states of components, which is part of the author's contribution. Finally, in section §3.3 is described related works in the division of grid application composition, deployment, monitoring and steering. In chapter §4 the focus is on formal methods for describing software applications, and how those methods can be used to verify the validity of the application construction. In section §4.1 formal methods are introduced, detailing the language used to describe grid applications in section §4.2 and illustrating verification techniques in section §4.3. Lastly, an outline is given on the issue of complexity in section §4.4 and solutions considered by the author in this research. In chapter §5 the researched approach in the formalism of behaviour of the Grid Component Model is described, one of the major contribution by the author. Starting by addressing formal specification of components in section §5.1, and continuing in section §5.2 by detailing the specification process involved when dealing with the environment of the grid application. In section §5.3 the focus is on the aspect

of dynamic reconfiguration of components, and the approach used to achieve it through model update. In chapter §6 it is illustrated the author's implementation of a prototype based on the research. The process involved in the design of the prototype is described in section §6.1, and how it was integrated in the related work of the Grid Development Environment in section §6.2. The features that this tool provides are described in section §6.3, and the testing against use cases is outlined in section §6.4. Finally, in chapter §7, are provided concluding remarks and identified possible future works.

Chapter 2

Integration

In recent times, software is growing in complexity in ways that seems to surpass people's ability to keep up with changes. We are not in fact talking about lines of code, calls to methods or loops in programs, but also about code spread between different locations, running in parallel and requiring a greater variety of external resources. Clouds, Grids, Service Oriented computing are just the most common examples, and in the near future the way we look at complexity will have to follow these developments. Furthermore, users of these systems expect constant new features to be added, problems to be addressed, and that the systems be as adaptive to their needs as possible. There has been a great amount of research in terms of component models [WS01], their structure [Ste99], reconfiguration (both static and dynamic) [BR00], even some autonomous approaches (a patent for dynamic software updates [Mar94] has also been filed). This topic can be very broad, and many of the cited researches overlap in one way or another, some trying to address a specific problem in a small context, some giving just a comprehensive overview of the issue without addressing any real type of scenario. We must not forget that any type of software system has to interact with its environment. From the common household appliance to the most sophisticated scientific computational systems, these software systems are generally developed to comfortably sit on top of some infrastructure. As the software develops,

so does the way it interacts with its environment, and it becomes more and more difficult to keep track of changes, since every environment may be affected by sudden and unplanned changes (e.g. power failures or human errors). This is why complex software is often designed to compensate in a way or another and adapt to these changes. Although these features are developed with the intent of compensating for any type of problem which might arise, often they do not consider each and every type of scenario, otherwise the complexity of the system would become unmanageable; but rather utilize the underlying structure of the framework the system is developed upon (such as a specific API), and give a fail-safe mode the software can revert to in case of malfunction. By keeping this in mind, we can see that this process cannot be foolproof, and often such a framework could lead to software still malfunctioning, although in a "safe" way. This is an underlying problem with the way the software and the framework it is developed upon are integrated. This can be considered trivial in the case of simple home applications (such as a word processor), but it becomes of great concern when scientific applications, or large scale systems in general, are involved. It becomes apparent that a formal approach to the development of such large scale infrastructures could come as a solution to the problem. Unfortunately it is often difficult to develop a framework with an underlying structure which allows for some kind of formal development, and some research has tried to address this point [KMWM03, Lin01], by using the well known planning properties of agents. The approach taken in this thesis is to instead integrate formal tools to a pre-existing framework, so that the approach can scale to other frameworks and be adapted to different kinds of software systems.

When deciding to solve a software (or hardware) problem using formal verification, it is common practice to choose one specification technique over another with regard to the best one that would fit the problem at hand, as well as of any software requirements needed to integrate the technique with other systems it may rely on or run in conjunction with. Furthermore, in some cases, more than one specification technique might be applicable or needed. At this point, integration of formalism and application becomes crucial for a successful output. But

while most research in the area is concentrated on integration from the point of view of interaction between different specification techniques, and between methods and tools still aimed at different techniques [EDD+04], the side of integration between formal specification and a 'system of tools' is often overlooked, as one is essentially built "ad-hoc", while the other is of a more combined approach. By 'system of tools' it is referred to a combined softwares which is composed of a number of different modular parts which fit together to perform a task, but which could evolve over time, often in a dynamic and distributed way, and perform the task in a different way. This is the case in system built from a model-based prospective, which is analysed in this research. While it is impossible to claim that a single concept for formal specification can be fitted to any component model, it is possible to say that an integration technique can be adapted to fit varying techniques of formal development to a number of abstract models typology. From this, it is possible to have a clear structure on how the integration can be fitted to different scenarios. In order to do this, the first step has to be to construct a proper abstraction of the component model, as well as one for the distributed execution. It is possible to then extract the formal development needed and fit in the parts to match the abstractions using common patterns to end up with a formal representation of the system. This approach is on the author's core contribution in this research.

2.1 Model Abstraction

In [JCK98], abstraction is defined as a process of elimination of irrelevant details in order to focus on the "essence of the problem at hand". Model abstraction is often associated with model checking, where a software system is too complex to be formally represented, so it is reduced in complexity in order to give a more simplified view. More precisely, by using model abstraction it is possible to reduce the number of states for a formal verification while preserving the structure and functionality of the original model. Although much research has been carried out in the field, [HL98] [Fra95] to name a few, it is often the case that such abstractions lose in one way

or another a part of the functionality of the original system which might be of importance. In order to minimize this loss, two parts of abstraction are needed, where instead of 'translating' the whole software system, it has been tried to map the parts which are relevant to our aim, and disregard the rest. In this research it is therefore referred to abstraction in terms of "reduction by abstraction", the idea of which was first introduced in [CGL94], and was created because of the need for reducing infinite transition systems to finite ones, so that the available specification languages and verification techniques would be able to cope with the complexity. The concept has been extended since to be applicable to other temporal logics [Kel94, CIY95, DGG97], but similarly to the research in [CC99] the abstraction needs to be extended to transition systems which are infinite, in order to take advantage of their characteristics; furthermore, as in this research it is considered the execution environment of the system, the abstraction should be extended to comprise this aspect. In this research, two abstractions have been modelled; the first depicts a view of the static part of the system, called the Component Model abstraction; while the second is a view of the system over time and through parallel executions, called the Distributed Execution Abstraction. Both are needed to have a complete picture of the system. While it is the case that the first, static, view has been researched and implemented with various successful techniques - mostly through mathematical abstractions for model checking [CGL94] - it is not sufficient without the dynamic insight that a Distributed Execution Abstraction can provide.

2.1.1 Component Model Abstraction

When dealing with abstraction of component models, it is easy to think of the procedure as similar to any other abstraction, and this is generally the case, especially in model checking scenarios. Some research has focused on taking components separately, and creating an abstraction for each of them [BJ08], having the drawback of not considering the interconnections between components, and their relation to the environment. Needless to say, the complexity of abstraction only of components taken separately can be still high, and it is just not practical

if we want to consider also the environment. While an in-depth abstraction of all the inner workings of each component in a component model is indeed essential to ensure correct execution of each component, this is not necessarily the case when dealing with the complete system; therefore only the outer workings of the component model are taken into account, i.e. the connectivity between components and components and resources. In this research it is devised a method through which an automata approach is designed to gather the necessary formalism, and a behavioural model can be applied.

2.1.2 Distributed Execution Abstraction

The research in abstractions of a software execution process, especially if distributed, is very slim, and often domain specific [DJ93], whereas the area of machine learning of environments and their behaviour [She94, KPP$^+$04, XZ08] is more developed. Unfortunately when dealing with large distributed systems the developer often falls into the assumption that any problem related to the execution of the software will be handled by the environment itself. Needless to say, this is not always the case, and a method to infer this abstraction is needed to complete the system's abstraction. In order to capture the distinctive nature of long term running distributed environments, an appropriate abstraction of the distributed execution of the system has to be constructed. Similarly to the abstraction for the component model, by utilizing an automata structure to infer the process of execution, it is possible to capture the state behaviour of components when running, as well as the state behaviour of any resource in the environment. The abstraction is built in a agent-like style, so that other parts of the environment which might be crucial to its abstraction can be analysed and added, in a process which is very similar to that in sequential or non-episodic agent environment. In these types of properties, the agent experiences the environment by dividing it into episodes; after each episode perception, the agent reacts with an action. This action though is not based only on the single perceived episode, but it also relates to previous episodes, giving the agent the potential to 'think ahead' when performing an action, as demonstrated in [TL02]; this process, although more complex

than simple episodic agent environments, can be more useful in the area of distributed execution, where planning ahead for future changes might prevent unforeseen failures.

2.2 Integrating Abstract Models

When talking about integration of model abstractions it is important to clarify the technique used to achieve it, as this determines whether the formal specification abstracted from the software system is a faithful enough representation of the concrete application and its environment. In this research, two abstractions have been identified, one for the component model, and the other for the distributed execution. When considering which approach to best combine the two together, it became clear that a manual process would be wasteful, but it also emphasized the fact that many steps involved are repetitive. Thanks to the clear specification of a component model's structure, the abstraction of its fundamental parts is greatly simplified. As there is no need for the creation of an abstract interpretation of data structures and the business code [CR94], the focus can be on parts like type of connections, hierarchical relationship, etc. It is easy to see that, for example, all broadcast connections (one to many) between interfaces of components in a system can be represented in a similar way, independently from how many components are in the relationship. Furthermore, this process is made even easier due to the innate properties of the formal language chosen, since a repetitive process can be easily duplicated and fitted into the formal specification. This process remind of similar procedures in image processing [Rus02], where patterns are used to identify similarities and associate them with a particular mathematical formula; it therefore evolved to consider pre-built patterns, called 'skeletons' [ACD+08], defined in the chosen Component Model, allowing for a more accurate representation of these sections of the model. A crucial part of the pattern creation is to ensure that it is written in the least complex manner. As the whole system if represented through a set of patterns, the aim has to be to try to lower the complexity for each pattern, as this, together with the way these patterns are assembled into a structure, could lead to reducing

the specification complexity.

Integration of formal methods in software developments can be challenging to accomplish, especially in today's complex systems. In this chapter it is described how Abstract Models methods can be used to achieve a faithful representation of a system, without having to resort to compromises, by choosing a technique which fits the requirements of the final aim. When having to deal with Grid systems, and specifically with their large and distributed nature, it is often easy to try to consider too many aspects in creating an abstract model. If the aim is to reduce the overall complexity of the integrated Abstract Models, it is important to clearly identifying the aspect most crucial when dynamically reconfiguring a Grid - i.e. the state behaviour of the Component Model, and its Distributed Execution; as well as shredding off parts that, while important to other approaches such as static model checking, are not important to the tackled scenario - such as data structures and business code.

Chapter 3

Grids and Component Models

The concept behind Grid computing is to utilize the processing power of computers when they are idle, and/or purpose-built clusters of workstations, as one big powerful supercomputer. To achieve this, low level Grid Infrastructures have been developed [FKNT02], often utilizing Component Models as their underlying structure [BFH03]. In this chapter it is analysed related work on Grids types and Component Models as a type of composition structure.

3.1 Grid types and structures

Between the various approaches to building long-lived and flexible Grid systems, the main ones are exhaustive and generic [CXDM04]. The first approach provides rich systems satisfying every service request from applications, its implementation consequently suffering from very high complexity. While in the second approach, it is represented only the basic set of services (minimal and essential) and thus overcome the complexity of the exhaustive approach. However, to achieve the full functionality of the system, it is essential to make this lightweight core platform reconfigurable and expandable. One of the possible solutions here is to identify and describe the basic set of features of the component model and to consider any other functions as pluggable components [TIG04] which can be brought on-line whenever necessary [GRSF04].

Establishing the theoretical foundations of the generic processes involved in designing and functioning of such Grid systems is highly important.

3.1.1 Classifications

Classifying different types of Grids can be controversial, there is in fact no clear definitions of classification of a Grid based on its core functionalities [Sto07]; however, it is common practice to make the most basic distinction between Computational Grids and Data Grids - the first being application centric and aiming at providing the highest processing power, while the second dealing with the sharing and management of large amounts of data, often focusing on storage and reliability rather than power [KF98]. Some other more subtle differentiations include [Zit07]:

- Networking Grids: where the main concern is on fault-tolerance during communication.
- Collaboration Grids: where the aim is to provide a platform for collaboration in distributed projects.
- Utility Grids: usually related to grids that make available specialized resources.

Other classifications deal with the size and location of Grids, such as with Cluster Grids (composed of a single localized cluster of machines), Enterprise Grids (where the machines are spread across multiple locations) and Global Grids (referring to a Grid widely spread and controlled by multiple organizations). With a similar terminology as above, Grids are also classified by complexity and conceptual models: Collaboration / Enterprise Grids refer to widely distributed grids characterized by business models; while Cluster Grids refer to static, high performance computing systems.

3.1.2 Grid Structure

As outlined in [FT05], the Grid Infrastructure must provide the following basic aspects:

- Resource modelling: insight on resources, their uses, availability, etc.
- Monitoring and notification: giving real time updates on the status of the application and the resources being used
- Allocation: ensuring that services are provided and requests are met
- Life-cycle: ensuring that resources are allocated for the life of the application
- Auditing: tracking usage of resources

Even though this is a very minimalistic list, it already gives an idea of the level of comprehensiveness which a generic grid must provide.

The Open Grid Service Architecture [Tal02], an architecture for a service-oriented grid computing environment for business and scientific use, expands on the capabilities in its documentation by adding:

- Infrastructure services
- Execution Management services
- Data services
- Resource Management services
- Security services
- Self-management services
- Information services

In the next section, it is analysed how a similar structure has been developed with a Component-based approach in the Grid Component Model (GCM), and what particular aspect of this development makes it an interesting approach in the realm of this research.

3.2 Component Models

Component models have been used in software engineering for years on a wide range of systems, most notably being Microsoft's COM [Box98] and ActiveX [Cor], and industrial standards such as Corba [OMG06] and JavaBeans [MHW03]. The main concept was created when the need to give software some form of abstract structure was presented [Mci68]; the initial idea behind software components was to take a similar approach as to hardware components, where the complexity of constructing machines was already been addressed by this method. The concept evolved and took different forms, but the fundamental idea of constructing software in basic building blocks - components - which could be arranged and connected together, remained. Most notably, in Component-based software engineering, a component is defined as a package which provides a set of functionalities. More precisely, as defined in [SGM02], the characteristic properties of components are: to be a unit of independent deployment and third-party composition, and to have no persistent state. This clearly gives a picture of a component that can be developed independently from any other that will form the final system (fundamental to the concept of re-usability), and that a component is essentially stateless (which becomes a core issue when dealing with Web services for example). The concept has evolved to include or inspire many others, like Object-oriented Architecture (where instead of on components, the focus is on modelling real world objects), Service-oriented Architecture (where a component becomes a service), and so forth. The idea of a component being stateless has also changed to adapt to different needs, mostly by assuming components to be inherently stateful. In frameworks like Corba [OMG06], we can see a similar description of components as to the one analysed in this research, where components are essentially black boxes in terms of functionality, but provide a defined set of interfaces to allow communication.

3.2.1 The Grid Component Model

Among various approaches to representing a component model specific attention is paid to the Fractal component model [BCS02]. The advantage of the Fractal framework is that it defines the structure of the components, gives a basic classification, and has a mathematical foundation, e.g., the Kell calculus [BS03]. The Fractal specification defines the basic (non-functional) controls which should be defined especially to enable dynamic reconfiguration of components, and a number of constraints on the interplay between functional and non-functional operations. The reconfiguration aspect in this case, is obtained by triggering appropriate actions on specific types of the components' interfaces. These explicit dynamic properties of the Fractal component model are particularly suitable for Grid systems and environments.

Fractal is a modular and extensible component model. The Fractal specification defines a set of notions characterizing this model, an API (Application Program Interface), and an ADL (Architecture Description Language).

Components are containers for some programming functionality; they are characterized by their *content* and the *membrane* that wraps them. The content of a component can be hidden (in which case it is simply a black box), or it can have some aspects of its inner functionality and structure revealed (grey boxes); and a component can be constituted by a system of some other components (referred as to sub-components). In the former case a component would be called *primitive* while the latter case represents a *composite* component. The membrane, or controller, controls the component. *Controllers* address non-functional aspects of the component.

Fractal is a multi-level specification. Depending on their conformance level, Fractal components can feature introspection and/or configuration. The *control* interfaces are used in the Fractal model to allow configuration (and reconfiguration), and are defined as *non-functional*. On the other hand, the functional interfaces of a component are associated with its functionalities. A *functional* interface can provide the required functionalities and it is called the *server* interface. Alternatively, a *client* interface requires some other functionalities.

Component interfaces are linked together by *bindings*. In the following, we will consider some primitive bindings, simple bindings transmitting invocations between the client interface and the connected server interface.

There are four controllers that have been already defined in Fractal (but others may be user-defined depending on the needs of the model):

- The *attribute controller* is used to configure a property within a component, when there is no need to take into consideration bindings of interfaces.
- The *binding controller* is used when the attribute controller is not applicable and actual binding/unbinding of interfaces is required.
- The *content controller* can be used to retrieve the representation of the *sub components* and add or remove them accordingly; note that if a sub component is *shared* by one or more other components, the scenario must be defined so that also these other components are taken into consideration.
- The *life cycle controller* allows to start and stop a component; it is used for dynamic reconfiguration so that all other controls can be applied safely to the component while the component is not in execution.

These are the basic controls which should be defined especially to be able to have dynamic reconfiguration of components.

The Fractal specification defines a number of constraints on the interplay between functional and non-functional operations:

- Content and binding control operations are only possible when the component is stopped.
- When stopped, a component does not emit invocations and must accept invocations through control interfaces; whether or not an invocation to a functional interface is possible is undefined.

The Grid Component model (GCM) [BCD+09] is an extension of Fractal built to accommodate requirements in distributed systems, in particular, those developed within and following the CoreGRID [oE] project. The GCM specification defines a set of notions characterising this model, an API (Application Program Interface), and an ADL (Architecture Description Language) [BHC+06]. In Fractal, when changing the bindings of a component, this component must be stopped (in other words, to avoid disruption to the system, when unplugging a component, such component must be stopped before severing its connections to other components); at the same time, invocation on controller interfaces must be enabled in order to send the stop signal to the component, making it de facto impossible to reconfigure the component controller. In GCM section 8.1 of [DD07], the life-cycle controller is extended allowing to separate partially the life-cycle states of the controller and of the content. When a component is functionally stopped (which corresponds to the stopped state of the Fractal specification), invocation on controller interfaces are enabled and the content of the component can be reconfigured. When a component is stopped, only the controllers necessary for configuration are still active (mainly binding, content, and lifecycle controllers), and the other components in the membrane can be reconfigured. It is possible to make use of these extended capabilities and monitor the changes in states of components.

The recent development of a Grid Integrated Development Environment (GIDE) based on the GCM specification [BGTI08] opens new possibilities for the dynamic reconfiguration scenario in large distributed systems. It is possible to take advantage of pre-built properties in the GIDE (namely the components' hierarchical composition, their API, and the monitoring of both components and resources) to form a basis for a reconfiguration framework which exploits the underlying properties of the specification language and deductive reasoning verification methods used in this research. We consider the monitoring specification of [BCS04] and the state information that can be retrieved through calls to the `LifeCycleController` interface (`getFcState` operation) for components, as well as other monitoring techniques for the environment.

Behaviour of states

The lifecycle of components in a component model is defined by states, allowed transitions and operations. As each component is such that it conforms to a set of defined states, it is possible to consider composite components (large components which are composites of primitive components and/or other composite components) as components that inherit the same properties and conform to state composition. In a system with multiple components in fact, the lifecycle of the whole system is defined by the relationships between the individual component lifecycles, and the state of each component is bound to the state of the components it relies on. Furthermore, the hierarchy of the system defines relationships where related components' lifecycles are linked: it is possible to define explicit semantics for guiding lifecycle transitions by using the component model itself, the ADL specification and the deployment information.

The basic lifecycle of components described in the GCM, and thus the resources being managed, can be retrieved at runtime by the use of the Component Monitoring and Resources Monitoring systems, built in the GIDE, through:

- components state calls (implemented by all component objects)
- specialised parameters monitoring for some specific components
- resources availability monitors
- metadata information

The state system is often restricted, in that it supports the deployment processes used by the framework and models only the deployment state of the system, not its operational characteristics. However, since each deployment component independently represents the state of the deployed resource which it is managing, the system as a whole must also represent a reasonable depiction of the overall state of many components.

The lifecycle of a component is assumed to behave as an automaton, whose states represent execution states of the component; it corresponds to an automaton with two states called

STARTED and **STOPPED**, where all the four possible transitions are allowed. It is however possible to define completely different lifecycle controller Java interfaces to use completely different automatons, or to define sub interfaces of this interface to define automatons based on this one, but with more states and more transitions [BBC+06]. A great number of component models in fact consider by default a number of substates to the most generic STARTED state, allowing for a deeper introspection on the behaviour of states of components (initialized, suspended, failed...).

3.3 The GCM: Composing, monitoring and steering

The Grid Component Model, a powerful framework for building Grid applications based on Fractal, provides a number of useful structural definitions for Grid developers as well as users. Its main features in fact include a strict foundation for hierarchical composition of applications, as well as structural designs for the monitoring and dynamic steering of such applications. In this section, the GCM framework is analysed, focusing on the aspect of reconfiguration, how it is achieved, and the problems which may arise during one.

3.3.1 Hierarchical composition

In the GCM, the hierarchical composition of the application is defined through the Architectural Description Language (ADL), which has the capabilities to describe components, connectors and configurations as well as the hierarchical structure of the system; however it is known that ADLs generally cannot provide sufficient insight into the post-deployment / runtime reconfiguration [MMHR04]. Although there is some research on how to extend the capabilities of ADLs to capture dynamic composition [PFT03], the simplest approach to surpass these restrictions is to rely on specific characteristics about the states of instantiated components (also known as 'live components') using standard runtime monitoring tools. It is possible to retrieve the specific state information (described in the previous section) as messages passed to the system

thus describing the runtime behaviour of states of the component. Similarly, the overall view of behaviour of states of the system of components and resources, describes the runtime behaviour of the environment.

3.3.2 Monitoring of components and resources

Grid systems are often very large and complex, and monitoring of the application and its resources becomes an essential tool [AAB+04]; the benefits range from optimization to failure detection and debugging. In the case of this research, monitoring of components and resources at runtime allows to recurrently pull state information of components and resources at any given moment. This is essential during a reconfiguration procedure in order to have a picture of the current overall state of the system and its environment. It is possible to create a direct mapping of the snapshot of the states taken during monitoring, to the state trees constructed during the formal specification process, effectively giving a starting point for the prediction of potential failures in the system - a crucial aspect which is lacking from other formal developments in the area [BHM05].

3.3.3 Dynamic reconfiguration in Grids and the GCM

In general, the initial configuration of GCM component is given by the description of the component using the GCM ADL.

From this first state, reconfiguration is obtained by triggering appropriate actions on the life-cycle, the binding, and the content control interfaces. A reconfiguration can be triggered by any component that has a reference to a correct non-functional interface.

Architecture

Let us now examine an overall picture of the architecture involved in the reconfiguration approach of this thesis, and where each part of a Grid system fits with the specification/verification scenario. Three main parts of the architecture can be identified: the primitive components, their

composition into composite components through the Architecture Description Language (ADL) file and the infrastructure (see Figure 3.1).

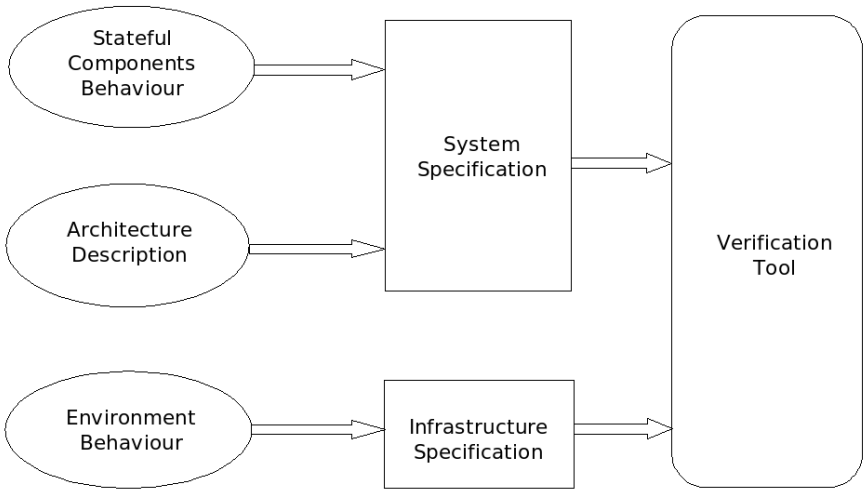

Figure 3.1: Architecture

The first two parts above are combined to deduce the *stateful component system behaviour* - a high-level behaviour distinct from the one of a single component, which is assumed to be already formally verified through other techniques being recently researched. The specification is partially given as an input by the developer in the case of resources, and partially automatically extrapolated using different sources, such as the ADL file and deployment file. The infrastructure is specified mainly according to the user's need, and following well-defined and accepted constrain such as those for safety, fairness, etc. [MP92] and in relation to the resources required and services provided. The formal specification derived through this process is a fusion of deontic and computation tree temporal logic, extended from the previous developments in [BBB+06], which is a suitable input format for the deductive reasoning tool. The properties to be specified and verified by these techniques are the ones which cannot possibly be considered when a system is specified in a static way, including but not limited to: presence of resources and services, availability of distributed components, etc.

In the classical approach to component behaviour specification, the term 'behaviour' refers to the component's inner functionality - if the component is supposed to calculate the factorial,

is it doing it correctly? When considering the stateful component system behaviour instead, it is taken into consideration a different aspect: we are looking for those requirements that will make the component 'behave correctly' in its environment. As a simple comparison, consider a parser which checks if all the libraries required by the component are present to calculate the factorial. Furthermore, what happens when we talk about a distributed system, where changes might be needed to be done at runtime? What if we require to replace a component, but the component we want to replace should not be stopped? These types of situations have been considered in this research while developing a specification procedure. It has been analysed the life cycle of a component and defined its states in a formal way so that they can be used in the system specification. Past developments within the GCM and other state aware grid systems [SK04] have been considered in order to define a set of states to be generated that would be monitored by specific software [GBT+07]. This lifecycle is restricted, in fact it only models the deployment state of the system (and, consequently, the transitions of its states during the lifecycle), not its operational characteristics. For example, once a component is in running state, it is available. On the other hand, the service may fail for other unforeseen circumstances (hence the need for a component monitoring system during runtime which will report a need for changes into the state behaviour specification).

In this chapter, some basic concepts and related works in the areas of Grid Computing and Component Models and their frameworks have been analysed, paying close attention to their benefits and/or shortcomings in the area of dynamic reconfiguration which is a core aspect in this research. The Grid Component Model (GCM) has been identified as the framework considered in this research, as well as analysing how the reconfiguration process is achieved and determining what are the aspect that make the GCM approach unsafe for the system.

Chapter 4

Formal Specification and Deductive Verification

In [BH95] it is argued that formal methods should follow the system engineering context for 'method', where it is given an underlying model of development, a language to express it, some steps to follow and some guidance on how to proceed between them. [Cro97] expands on this by identifying six phases:

1. Characterization: understanding of the application and its domain
2. Modelling: mathematical representation of the overall application
3. Specification: logical formalization of relevant parts of the application
4. Analysis: validation of the specification
5. Documentation: record assumptions and motivate decisions
6. Maintenance and Generalization: modify the specification as required

The aim of this chapter is to address the central phases identified above. Related works in formal methods for specification will be identified, describing the languages that were adopted,

and the verification process used to prove properties and invariants of the system, including author's contributions in the extension of these languages.

4.1 Formal Methods

"The term Formal Methods refers to the use of techniques from logic and discrete mathematics in the specification, design, and construction of computer systems and software" [Cro97]. In this section it is analysed how formalism can be used to approach the problem of formally specifing the Grid Component Model and verifing certain properties. Specific areas of the Grid system will be focused on, as defined previously in the Model Abstractions, and given an insight on the reasons for choosing one technique over the other.

4.1.1 Formalism in software development

Formal approaches to software development are widely used and researched [All97, BBC05, WK02]. Unfortunately, the habit of systematically embedding formalism in implementation is not. It is in fact common to find formal approaches which complement software developments [CW96]. It may be the case that the need for a formal approach is only discovered at a late stage of developments, or that it might just not be fundamental and therefore be used in a second stage. In most cases, the job is left to pre-existing tools which can create a formal specification of a system (usually through some ad-hoc implementation) and others to verify their properties. As a problem presents itself in a system where the best solution is through some formalism, one of these tools is usually chosen to perform these tasks. In the case of this research, the tools were just not present. It was therefore developed an approach to software formalism, and in particular the formalism of Grid Component Models that is novel and non-restrictive. The author is not aware of any other developments which place so much importance on the formal specification of grid environments. Furthermore, the integration process has been generalized so that this type of formalism can be adapted to a number of other component

models, and does not only have to be restricted to Grids, but possibly open to other types of distributed systems, large scale and long running software, and parallel computing applications.

4.1.2 GCM Approach

The Grid Component Model exposes a clear structure and insight on the functionality of the developed Grid system. Thanks to the definitions in [CCH+08] and their formal fundamentals [DD07], it is possible to deduce what formal method to use in order to ensure that the needs in critical assessments, assurance consideration and architectural characteristics [Cro97] are met. First it is essential to focus on the GCM components' definitions, where each deployed component MUST expose a state resource property, which implements the Component's Monitoring capability. To satisfy this requirement, a deployment component must contain **States** and **State Transition** elements. Additionally, a deployment component may include additional information as an opaque quantity that an external consumer may be able to process. The **Component Status** property will be exposed by every component object of a system.
These properties can be defined in the XML based system architecture as:

```
<ComponentStatus>
<State>UndefinedState | InstantiatedState | InitializedState |
RunningState | FailedState | TerminatedState</State>
<LifecycleTransition>StateTransition</LifecycleTransition>
</ComponentStatus>
```

where:

Element	Description
InstantiatedState	State representing the presence of a component instance.
InitializedState	State in which a component has been properly initialized.
RunningState	Operational state.
FailedState	State in which the component has failed either a lifecycle operation or its operation has failed.
TerminatedState	State in which a component instance has been terminated.

As the failed state may have been arrived at due to failures during many parts of the lifecycle, it is RECOMMENDED that the component take action to ensure the services of the resources are not available while in this state, particularly if the transition occurred from the running state.

Similarly, it is possible to map the state of resources and monitor changes through state change notifications fired by resource monitoring software implemented in the GIDE.

4.1.3 Agents

While speaking of software formalism for components and resources, it is easy to see the connection with the agent realm. Of course, a direct comparison between components and agents should be avoided, as in components we clearly do not see a sign of automated reasoning which we would find in agents [Fon93]. On the other hand, agent-oriented conceptual modelling has been thoroughly researched [VGK04] and has also seen some implementation [SBS09]. Agents have the capability of triggering and responding to actions, but only the latter is present in components, as they are required only to react to messages passed through their interface bindings (although there might be some cases in which they are required to perform otherwise). It is also the case for the reverse, where components are used to construct agents [KMWM03]. Another aspect in which agents are connected to the approaches described in this thesis, is the way in which the formal specification process is described in this research. It is easy to

see how the automata approach used behaves in a similar fashion to agents when building the specification, reacting to the scenario presented and making decisions on how to construct in a formal way the interaction between components and resources.

4.1.4 Model checking vs deductive reasoning

For the specification of behaviour, this research uses a rich temporal framework [Eme90] with subsequent application of either model checking or deductive reasoning as a verification technique. In [Cla97], Clarke defines Model Checking as an "automatic technique for verifying finite-state concurrent systems". In simpler terms, when a system is built to perform a specific task, with a specific set of properties to satisfy, model checking is used to test whether those properties are satisfied. It is commonplace to use a model of a system to automatically test whether it meets the specification containing safety requirements. The technique has already been tested in various scenarios [CFJV05], one particular application of this method has also been applied in similar circumstances [BHM05] to verify the inner behaviour of components (in other words, to ensure that the components perform the calculations they are supposed to); this is a powerful and well-established technique which allows to incorporate a number of algorithms and tools to deal even with the famous state explosion problem. However, when applied to a component system, it has one significant drawback, namely it has an explorative nature and it cannot efficiently handle infinite state systems [Eme08] (i.e. non-terminating systems); in fact, model checking is used to take "snapshots" of various static states of a system, and quickly verify them, but when considering a long running system - possible even infinite - it is easy to understand that this procedure becomes often infeasible. Although there has been much research on obviating this problem [And94, VAHL02], solutions are often restricted in solely the realm of cleverly designed abstractions [CDG01]. As a consequence, model checking has troubles considering the environment in which a component system has been developed. At the same time, in building a large scale distributed system, it is not possible to afford any

more not to take into consideration the entire infrastructure, as it has been extensively analysed throughout previous research [BBGH08]. Deductive methods, on the other hand, can deal with such large or even infinite state systems - as the technique has been developed precisely to solve this problem - and furthermore can be applied to reconfiguration scenarios, where we must consider future system states as a whole, and taking a series of "snapshots" would just be impractical. A resolution based verification framework for the fusion of temporal and deontic logics will be outline later in this chapter. In [ZHD08] the original resolution method for CTL [Bol00] has been improved by making the set of resolution rules more effective. This means that since in this system there is no interaction between the normative and temporal dimensions it is reasonable to take this improved set of resolution rules coping with the temporal setting instead of the one initially considered in [BB07] thus obtaining a more efficient resolution system. The correctness of the system follows from the correctness argument for both parts - temporal (as these new developments in [ZHD08] guarantee the correctness) and deontic (as shown in [BB07]).

4.2 Languages

The notations which will be analysed in the next section have been chosen while having in mind the overall task that was set out to be achieved, developing an environment-aware dynamic verification tool. The requirements would therefore have to be:

- To achieve a level of expressiveness hight enough to describe a stateful grid application as well as its environment.
- To allow to consider all the different paths the application might take in its lifetime (which could be infinite)
- To reduce the aspects which can be overly descriptive in order to keep complexity to a minimum.

- To have feature that would ease an automated reconfiguration of the specification produced.

While existing notations can provide all of the above points separately, there is no language capable of combining them into a suitable notation system. In fact, different types of logic languages are used to formally specify different types of systems, and the choice is left to the researcher to select one depending on the level of expressiveness required for the system in question; the advantages of choosing one over the other is simple to identify thanks to the fact that each of them has rigorous mathematical definitions. As this research had to describe a system which would possibly run forever, and may be subject to changes over time, it was logic to begin by looking into the realm of temporal logic, finally identifying branching time logic ECTL$^+$ as the best suited for this approach. The defining characteristic that led to this choice are, among others, the ability to integrate well in component based scenarios, adaptability in light of a possible reconfiguration as well as interoperability with the infrastructure. However, the fact that a Grid system might be dynamically reconfigured at runtime, posed some concerns on the level of expressiveness of ECTL$^+$, and its inability to describe a model update of its state tree. To allow this, ECTL$^+$ was extended with Deontic modalities ECTL^+_D. These techniques can all be unified under their normal form, and the final extended logic utilized for this research falls under the SNF$^D_{\mathrm{CTL}}$language, which is the final result contribution for this part of the thesis. It is now possible to construct a specification using the SNF$^D_{\mathrm{CTL}}$language which solves the shortfall of ECTL$^+$ in the area of dynamic re-configurability of its tree structure.

4.2.1 ECTL$^+$

The language ECTL$^+$ is an extension of linear-time temporal logic, to incorporate the notion of branching time. Furthermore, it expands on the expressiveness of its predecessor ECTL allowing nested temporal operators, in order to express boolean combination of fairness properties or

temporal operators when applied to path quantifiers [BB06]. This language extends linear-time logic temporal operators $\Box$ (always), $\Diamond$ (sometime), $\bigcirc$ (next time), $\mathcal{U}$ (until) and $\mathcal{W}$ (unless), with path quantifiers **A** (for any future paths) and **E** (for some future path). Similarly to CTL we have also, *state* (S) and *path* (P) formulae, such that well formed formulae are state formulae. These classes of formulae are inductively defined below (where C is a formula of classical propositional logic)

$$\begin{array}{ll} S ::= & C|S \wedge S|S \vee S|S \Rightarrow S|\neg S|\mathbf{A}P|\mathbf{E}P \\ P ::= & P \wedge P|P \vee P|P \Rightarrow P|\neg P| \\ & \Box S|\Diamond S|\bigcirc S|S\,\mathcal{U}\,S|S\,\mathcal{W}\,S|\,\Box\Diamond S|\Diamond\,\Box S \end{array}$$

Table 4.1: ECTL$^+$ *state* and *path* formulae

Underlying Tree Structures. Assuming familiarity of the reader with the basic tree structure concepts, let us continue with the presentation of the ECTL$^+$ language by the introduction of the notation utilized as described in [BB06].

Definition 1 (Tree) *A* tree, $\mathcal{T}$, *is a pair* (S, R), *where* S *is a set of states and* $R \subseteq S \times S$ *is a relation between states of* S *such that (a)* $s_0 \in S$ *is a unique root node (b)for every* $s_i \in S$ *there exists* $s_j \in S$ *such that* $R(s_i, s_j)$*; and (c) for every* $s_i,\ s_j,\ s_k \in S$, *if* $R(s_i, s_k)$ *and* $R(s_j, s_k)$ *then* $s_i = s_j$.

By χ_{s_i} it is possible to abbreviate a path departing from s_i. A path χ_{s_0} is called a *fullpath*. Let X be a family of all fullpaths of $\mathcal{T}$. Given a path χ_{s_i} and a state $s_j \in \chi_{s_i},\ (i < j)$ we term a finite subsequence $[s_i, s_j] = s_i, s_{i+1}, \ldots, s_j$ of χ_{s_i} *a prefix* of a path χ_{s_i} and an infinite sub-sequence $s_j, s_{j+1}, s_{j+2}, \ldots$ of χ_{s_i} *a suffix* of a path χ_{s_i} abbreviated $Suf(\chi_{s_i}, s_j)$.

Assuming that the underlying trees are countable ω-trees, i.e. any fullpath $\chi \in X$ is isomorphic to natural numbers and every state $s_i \in S$ has a countable number of successors.

Definition 2 (Countable ω-tree) *A* countable ω*-tree,* $\mathcal{T}_\omega$, *is a tree* (S, R) *with the family of all fullpaths,* X, *which satisfies the following conditions:*

- *each fullpath* $\chi \in X$ *is isomorphic to natural numbers;*

- *every state $s_i \in S$ has a countable number of successors.*

Definition 3 (Branching degree of a state) *Now it is possible to define the formal syntax and semantics for ECTL$^+$. A well-formed ECTL$^+$ formula is interpreted in a structure $\mathcal{M} = \langle S, R, s_0, X, L\rangle$, where (S, R) is a countable ω tree with a root s_0, X is a set of all fullpaths and L is an interpretation function mapping atomic propositional symbols to truth values at each state and the following condition is satisfied:*

- *X is R-generable [Eme90], i.e. for every state $s_i \in S$, there exists $\chi_j \in X$ such that $s_i \in \chi_j$, and for every sequence $\chi_j = s_0, s_1, s_2, \ldots$, the following is true: $\chi_j \in X$ if, and only if, for every i, $R(s_i, s_{i+1})$.*

- *a tree (S, R) is of at most countable branching.*

4.2.2 $\mathrm{SNF}_{\mathrm{CTL}}$

The language of a normal form, $\mathrm{SNF}_{\mathrm{CTL}}$ developed for a number of branching-time logics, CTL [Bol00, BF99], ECTL [Bol03] and ECTL$^+$ [BB06], uses the same language as ECTL$^+$ (defined above) without the $\mathcal{U}$ (until) and $\mathcal{W}$ (unless) operators, and is extended by the use of indices.

In the definition in [BB06] of an $\mathrm{SNF}_{\mathrm{CTL}}$ model structure $\mathcal{M}$ the set of fullpaths X is R-generable. Therefore, following [Eme90], it it is suffix, fusion and limit closed.

Syntax. First, begin by fixing a countable set, $Prop = x, y, z, \ldots$, of atomic propositions. The core idea of $\mathrm{SNF}_{\mathrm{CTL}}$ is to represent temporal information in the following three types of constraints. *Initial constraints* represent information relevant to the initial moment of time, the root of the computation tree. *Step constraints* indicate what will happen at the successor state(s) given that some conditions are satisfied 'now'. Finally, *Sometime constraints* keep track on any eventuality, again, given that some conditions are satisfied 'now'. Additionally, to enable sound reasoning within a specific path context during the verification, it is necessary to incorporate indices.

Indices. The language for indices is based on the set of terms

$\mathsf{IND} = \{\langle \mathsf{f}\rangle,\ \langle \mathsf{g}\rangle,\ \langle \mathsf{h}\rangle,\ \langle LC(\mathsf{f})\rangle,\ \langle LC(\mathsf{g})\rangle,\ \langle LC(\mathsf{h})\rangle\ \ldots\}$, where f, g, $\mathsf{h}\ldots$ denote constants. Thus, $\mathbf{E}A_{\langle \mathsf{f}\rangle}$ means that A holds on some path labelled as $\langle \mathsf{f}\rangle$. A designated type of indices in $\mathrm{SNF_{CTL}}$ are indices $\langle \mathsf{LC(ind)}\rangle$ which represent a limit closure of prefixes associated with $\langle \mathsf{ind}\rangle$. All Formulae of $\mathrm{SNF_{CTL}}$ of the type $P \Rightarrow \mathbf{E}\bigcirc Q$ or $P \Rightarrow \mathbf{E}\diamondsuit Q$, where Q is a purely classical expression, are labeled with some index.

Definition 4 (Separated Normal Form $\mathbf{SNF_{CTL}}$) *A set of* $\mathrm{SNF_{CTL}}$ *clauses is a set of Formulae* $\mathbf{A}\,\Box\,[\bigwedge_i (P_i \Rightarrow F_i)]$ *where each of the* clauses $P_i \Rightarrow F_i$ *is further restricted as below, each* $\alpha_j, \alpha_p, \alpha_t, \alpha_v,\ \beta_i, \beta_m, \beta_r$ *or* γ *is a literal,* **true** *or* **false** *and* $\langle \mathsf{ind}\rangle \in \mathsf{IND}$ *is some index.*

$\mathbf{start}$	$\Rightarrow$	$\bigvee_{i=1}^{k} \beta_i$	an initial clause
$\bigwedge_{j=1}^{l} \alpha_j$	$\Rightarrow$	$\mathbf{A}\bigcirc [\bigvee_{m=1}^{n} \beta_m]$	an $\mathbf{A}$ step clause
$\bigwedge_{p=1}^{q} \alpha_p$	$\Rightarrow$	$\mathbf{E}\bigcirc [\bigvee_{r=1}^{s} \beta_r]_{\langle \mathsf{ind}\rangle}$	an $\mathbf{E}$ step clause
$\bigwedge_{t=1}^{u} \alpha_t$	$\Rightarrow$	$\mathbf{A}\diamondsuit\gamma$	an $\mathbf{A}$ sometime clause
$\bigwedge_{v=1}^{w} \alpha_v$	$\Rightarrow$	$\mathbf{E}\diamondsuit\gamma_{\langle \mathsf{LC(ind)}\rangle}$	an $\mathbf{E}$ sometime clause

Table 4.2: $\mathrm{SNF_{CTL}}$ clauses

Interpreting $\mathrm{SNF_{CTL}}$

A relation $\models$ is defined which evaluates the $\mathrm{SNF_{CTL}}$ clauses at a state s_i in a model $\mathcal{M}$. The evaluation of the classical connectives in the states is standard. Below it is represented the evaluation of the temporal operators and path quantifiers.

$\langle \mathcal{M}, s_i\rangle$	$\models \mathbf{A}B$	iff	for each $\chi_{s_i}, \langle \mathcal{M}, \chi_{s_i}\rangle \models B$.
$\langle \mathcal{M}, s_i\rangle$	$\models \mathbf{E}B$	iff	there exists χ_{s_i} such that $\langle \mathcal{M}, \chi_{s_i}\rangle \models B$.
$\langle \mathcal{M}, \chi_{s_i}\rangle$	$\models \Box B$	iff	for each $s_j \in \chi_{s_i}$, if $i \leq j$ then $\langle \mathcal{M}, Suf(\chi_{s_i}, s_j)\rangle \models B$.
$\langle \mathcal{M}, \chi_{s_i}\rangle$	$\models \diamondsuit B$	iff	there exists $s_j \in \chi_{s_i}$ such that $i \leq j$ and $\langle \mathcal{M}, Suf(\chi_{s_i}, s_j)\rangle \models B$.
$\langle \mathcal{M}, \chi_{s_i}\rangle$	$\models \bigcirc B$	iff	$\langle \mathcal{M}, Suf(\chi_{s_i}, s_{i+1})\rangle \models B$.

Table 4.3: $\mathrm{SNF_{CTL}}$ evaluation of the temporal operators and path quantifiers

Definition 5 (Satisfiability, validity) *An* $\mathrm{SNF_{CTL}}$ *clause, C, is satisfiable if, and only if, there exists a model* $\mathcal{M}$ *such that* $\langle \mathcal{M}, s_0 \rangle \models C$. *An* $\mathrm{SNF_{CTL}}$ *clause, C, is valid if, and only if, it is satisfied in every possible model.*

An initial $\mathrm{SNF_{CTL}}$ clause, $\mathbf{start} \Rightarrow F$, is understood as "$F$ is satisfied at the initial state of some model $\mathcal{M}$". Any other $\mathrm{SNF_{CTL}}$ clause is interpreted taking also into account that it occurs in the scope of $\mathbf{A}\,\Box$.

Thus, a clause $\mathbf{A}\,\Box(x \Rightarrow \mathbf{A}\bigcirc p)$ is interpreted as "*for any fullpath* χ *and any state* $s_i \in \chi$ $(0 \leq i)$*, if* x *is satisfied at a state* s_i *then* p *must be satisfied at the moment, next to* s_i*, along each path which starts from* s_i".

Next, a clause $\mathbf{A}\,\Box(x \Rightarrow \mathbf{E}\bigcirc q_{\langle \mathsf{ind} \rangle})$ is interpreted as "*for any fullpath* χ *and any state* $s_i \in \chi$ $(0 \leq i)$*, if* x *is satisfied at a state* s_i *then* q *must be satisfied at the moment, next to* s_i*, along a path which starts from* s_i *and which is associated with* ind". Speaking informally, it is possible to interpret $\mathbf{A}\,\Box(x \Rightarrow \mathbf{E}\bigcirc q_{\langle \mathsf{ind} \rangle})$ such that given a state in a model which satisfies x (the left hand side of the clause), the label, ind, indicates the direction, in which the successor state which satisfies q can be reached (see similar developments in the construction of logic DCTL$*$ [HT87]).

Finally, it is important to point out that the interpretation of an LC index corresponds to the concept of a linear interpretation [Wol95].

Note that in the full $\mathrm{ECTL^{+}}$ language the standard 'until' ($\mathcal{U}$) and 'unless' ($\mathcal{W}$) operators are used:

$\langle \mathcal{M}, \chi_{s_i} \rangle \models A\,\mathcal{U}\,B$ iff there exists $s_j \in \chi_{s_i}$ such that $i \leq j$ and $\langle \mathcal{M}, Suf(\chi_{s_i}, s_j) \rangle \models B$

and for each $s_k \in \chi_{s_i}$, if $i \leq k < j$ then $\langle \mathcal{M}, Suf(\chi_{s_i}, s_k) \rangle \models A$ and $A\,\mathcal{W}\,B = \Box A \vee A\,\mathcal{U}\,B$

In the $\mathrm{SNF_{CTL}}$ these operators are defined via the basic set of $\mathrm{SNF_{CTL}}$ operators [Bol00]. The rules considered for the translation to $\mathrm{SNF_{CTL}}$ are the removal of $\mathbf{E}\,\mathcal{W}$ and $\mathbf{A}\,\mathcal{W}$, for the full procedure refer to [BB06].

4.2.3 ECTL_D^+

In this section it is introduced the temporal specification framework, which is based upon the language of the normal form ($\mathrm{SNF}_{\mathrm{CTL}}$) defined initially in [Bol00] as the underlying language for the clausal resolution method for the computation tree logic CTL. Here this setting is extended to capture a fusion of the logic ECTL+ (extended CTL, [Eme90]) and the deontic logic [LW06]. Thus, let us first start with the introduction of this expressive framework and then show how $\mathrm{SNF}_{\mathrm{CTL}}$ can be extended to TDS (temporal deontic specifications) so that any formula of ECTL_D^+ can be translated into a corresponding TDS, which preserves satisfiability.

Definition 6 (Temporal Deontic Specification - TDS) *TDS is a tuple* $\langle In, St, Ev, N, Lit\rangle$ *where* In *is the set of initial constraints,* St *is the set of step constraints,* Ev *is the set of eventuality constraints,* N *is a set of normative expressions, and* Lit *is the set of literal constraints, i.e. formulae that are globally true. The structure of these constraints called* clauses*, is defined below where each* α_i*,* β_m*,* γ *or* l_e *is a literal,* **true** *or* **false***,* d_e *is either a literal or a modal literal involving the* $\mathcal{O}$ *or* $\mathcal{P}$ *operators,* $\langle \mathsf{ind}\rangle \in \mathsf{IND}$ *is some index, and the clauses are supposed to be in the scope of the* $\mathbf{A}\,\Box$ *modality.*

start	$\Rightarrow$	$\bigvee_{i=1}^{k} \beta_i$	(In)	$\bigwedge_{i=1}^{k} \alpha_i$	$\Rightarrow$	$\mathbf{A}\Diamond\gamma$	(Ev **A**)
$\bigwedge_{i=1}^{k} \alpha_i$	$\Rightarrow$	$\mathbf{A}\bigcirc [\bigvee_{m=1}^{n} \beta_m]$	(St **A**)	$\bigwedge_{i=1}^{k} \alpha_i$	$\Rightarrow$	$\mathbf{E}\Diamond\gamma_{\langle \mathsf{LC(ind)}\rangle}$	(Ev **E**)
$\bigwedge_{i=1}^{k} \alpha_i$	$\Rightarrow$	$\mathbf{E}\bigcirc [\bigvee_{m=1}^{n} \beta_m]_{\langle \mathsf{ind}\rangle}$	(St **E**)	**true**	$\Rightarrow$	$\bigvee_{e=1}^{n} d_e$	(D)
				true	$\Rightarrow$	$\bigvee_{e=1}^{n} l_e$	(Lit)

Table 4.4: TDS clauses

In the rest of the thesis, to simplify reading, the prefix $\mathbf{A}\,\Box$, common for all TDS clauses, will be omitted.

The language of ECTL_D^+ follows the definitions of ECTL^+ above, and is extended by deontic modalities: assume a set $Ag = \{a, b, c \ldots\}$ of agents (processes), which are associated with deontic modalities $\mathcal{O}_a(\varphi)$ read as 'φ is obligatory for an agent a' and $\mathcal{P}_a(\varphi)$ read as 'φ is

permitted for an agent a'.

In the syntax of ECTL$_D^+$ are distinguished *state* (S) and *path* (P) formulae, such that S are well formed formulae. These classes of formulae are inductively defined below (where C is a formula of classical propositional logic)

$S ::= C|S \wedge S|S \vee S|S \Rightarrow S|\neg S|\mathbf{A}P|\mathbf{E}P|\mathcal{P}_a S|\mathcal{O}_a S$
$P ::= P \wedge P|P \vee P|P \Rightarrow P|\neg P|\ \Box S|\diamondsuit S|\bigcirc S|S\,\mathcal{U}\,S|$
$S\,\mathcal{W}\,S|\ \Box\diamondsuit S|\diamondsuit\ \Box S$

Table 4.5: ECTL$_D^+$ *state* and *path* formulae

Definition 7 (literal, deontic literal) *A* literal *is either p, or $\neg p$ where p is a proposition. A* deontic literal *is either $\mathcal{O}_i l$, $\neg\mathcal{O}_i l$, $\mathcal{P}_i l$, $\neg\mathcal{P}_i l$ where l is a literal and $i \in Ag$.*

ECTL$_D^+$ **Semantics.** For the interpretation of deontic operators, it is introduced a binary agent accessibility relation.

Definition 8 (Deontic Accessibility Relation) *Given a total countable tree $\tau_\omega = (S, \leq)$, a binary agent accessibility relation $D_i \subseteq S \times S$, for each agent $i \in Ag$, satisfies the following properties: it is* serial *(for any $k \in S$, there exists $l \in S$ such that $D_i(k,l)$),* transitive *(for any $k,l,m \in S$, if $D_i(k,l)$ and $D_i(l,m)$ then $D_i(k,m)$), and* Euclidian *(for any $k,l,m \in S$, if $D_i(k,l)$ and $D_i(k,m)$ then $D_i(l,m)$).*

Let $(S, \leq)$ be a total countable ω-tree with a root s_0, X be a set of all fullpaths, $L : S \times Prop \longrightarrow \{\mathbf{true}, \mathbf{false}\}$ be an interpretation function mapping atomic propositional symbols to truth values at each state, and every $R_i \subseteq S \times S$ $(i \in 1, \ldots, n)$ be an agent accessibility relation defined as in Def 8. Now a model structure for interpretation of ECTL$_D^+$ formulae is $\mathcal{M} = \langle S, \leq, s_0, X, L, D_1, \ldots, D_n \rangle$.

Recalling that, since the underlying tree structures are R-generable, they are suffix, fusion and limit closed [Eme90], below it is defined a relation '$\models$', which evaluates well-formed ECTL$_D^+$ formulae at a state s_m in a model $\mathcal{M}$.

$\langle \mathcal{M}, s_m \rangle$	$\models p$	iff	$p \in L(s_m)$, for $p \in Prop$
$\langle \mathcal{M}, s_m \rangle$	$\models \mathbf{A}B$	iff	for each χ_{s_m}, $\langle \mathcal{M}, \chi_{s_m} \rangle \models B$
$\langle \mathcal{M}, s_m \rangle$	$\models \mathbf{E}B$	iff	there exists χ_{s_m} such that $\langle \mathcal{M}, \chi_{s_m} \rangle \models B$
$\langle \mathcal{M}, \chi_{s_m} \rangle$	$\models A$	iff	$\langle \mathcal{M}, s_m \rangle \models A$, for state formula A
$\langle \mathcal{M}, \chi_{s_m} \rangle$	$\models \Box B$	iff	for each $s_n \in \chi_{s_m}$, if $m \leq n$ then $\langle \mathcal{M}, Suf(\chi_{s_m}, s_n) \rangle \models B$
$\langle \mathcal{M}, \chi_{s_m} \rangle$	$\models \bigcirc B$	iff	$\langle \mathcal{M}, Suf(\chi_{s_m}, s_{m+1}) \rangle \models B$
$\langle \mathcal{M}, \chi_{s_m} \rangle$	$\models A \,\mathcal{U}\, B$	iff	there exists $s_n \in \chi_{s_m}$ such that $m \leq n$ and $\langle \mathcal{M}, Suf(\chi_{s_m}, s_n) \rangle \models B$ and for each $s_k \in \chi_{s_m}$, if $m \leq k < n$ then $\langle \mathcal{M}, Suf(\chi_{s_m}, s_k) \rangle \models A$
$\langle \mathcal{M}, \chi_{s_m} \rangle$	$\models A \,\mathcal{W}\, B$	iff	$\langle \mathcal{M}, \chi_{s_m} \rangle \models \Box A$ or $\langle \mathcal{M}, \chi_{s_m} \rangle \models A \,\mathcal{U}\, B$
$\langle \mathcal{M}, s_m \rangle$	$\models \mathcal{O}_a B$	iff	for each $s_n \in S$, if $D_a(m, n)$ then $\langle \mathcal{M}, s_n \rangle \models B$
$\langle \mathcal{M}, s_m \rangle$	$\models \mathcal{P}_a B$	iff	there exists $s_n \in S$, such that $D_a(m, n)$ and $\langle \mathcal{M}, s_n \rangle \models B$

Table 4.6: Well-formed ECTL^+_D formulae

Definition 9 (Satisfiability) *A well-formed* ECTL^+_D *formula, B, is satisfiable if, and only if, there exists a model $\mathcal{M}$ such that $\langle \mathcal{M}, s_0 \rangle \models B$.*

Definition 10 (Validity) *A well-formed* ECTL^+_D *formula, B, is valid if, and only if, it is satisfied in every possible model.*

4.2.4 $\text{SNF}^D_{\text{CTL}}$

To define a concept of propositional deontic temporal specification, the normal form defined for the logic ECTL^+, SNF_{CTL}, which was developed in [Bol00, BF99] is extended. Recall that the core idea of the normal form is to extract from a given formula the following three types of constraints. *Initial constraints* represent information relevant to the initial moment of time, the root of a tree. *Step constraints* indicate what will happen at the successor state(s) given that some conditions are satisfied 'now'. Finally, *sometime constraints* keep track on any eventuality, again, given that some conditions are satisfied 'now'.

The $\text{SNF}^D_{\text{CTL}}$ language is obtained from the ECTL^+_D language by omitting the $\mathcal{U}$ and $\mathcal{W}$

operators, and adding classically defined constants **true** and **false**, and a new operator, **start** ('at the initial moment of time') defined as

$\langle \mathcal{M}, s_i \rangle \models \mathbf{start} \quad \text{iff} \quad i = 0.$

Similarly to $\mathrm{SNF}_{\mathrm{CTL}}$, the language for indices is incorporated, based on the set of terms $\mathsf{IND} = \{\langle \mathsf{f} \rangle, \langle \mathsf{g} \rangle, \langle \mathsf{h} \rangle, \langle LC(\mathsf{f}) \rangle, \langle LC(\mathsf{g}) \rangle, \langle LC(\mathsf{h}) \rangle \ \ldots\}$, where f, g, $\mathsf{h}\ldots$ denote constants. Thus, $\mathbf{E}A_{\langle \mathsf{f} \rangle}$ means that A holds on some path labelled as $\langle \mathsf{f} \rangle$. All formulae of $\mathrm{SNF}_{\mathrm{CTL}}$ of the type $P \Rightarrow \mathbf{E}\bigcirc Q$ or $P \Rightarrow \mathbf{E}\diamondsuit Q$, where Q is a purely classical expression, are labelled with some index. Also, as in $\mathrm{SNF}_{\mathrm{CTL}}$, the rules considered for the translation to $\mathrm{SNF}^{D}_{\mathrm{CTL}}$ are the removal of $\mathbf{E}\,\mathcal{W}$ and $\mathbf{A}\,\mathcal{W}$, in a similar fashion as to [BB06].

4.2.5 Automata based approach to Formal Specification

In building the specification protocol, the well known automata constructions are followed. A simple finite state automaton on finite strings is considered, and a set of specification "patterns" is applied. The automata is used for the creation of labels defining various states in which the considered components and resources can be, and the derived model is then directly specified in a formal manner.

Automata Construction

A finite automaton on finite words is utilized in constructing the specification protocol. Let Σ be a finite alphabet. A finite word over Σ is an element of $\Sigma^{\star}$.

Definition 11 (Finite Word Automaton) *A finite word automaton [Wol01], A, is a tuple* $A = (\Sigma, Q, Q_i, Q_f, \Delta)$ *where* Σ *is a finite alphabet,* Q *is a set of states,* $Q_i \subseteq Q$ *is a set of initial states,* $Q_f \subseteq Q$ *is a set of accepting states, and* $\Delta : Q \times \Sigma \longrightarrow 2^S$ *is a transition function.*

A run, R, of A over a word $w = a_1, a_2, \ldots, a_n - 1,\ w \in \Sigma^{\star}$ is abbreviated as R_w and it is a sequence of states $s_1, s_2, \ldots, s_n$ such that for any i, $(0 \leq i < n)$, $s_{i+1} \in \Delta(s_i, a_i)$. A run, $R = s_1, s_2, \ldots, s_n$, is successful if $s_1 \in Q_i$ and $s_n \in Q_f$. An automaton A accepts a word w if

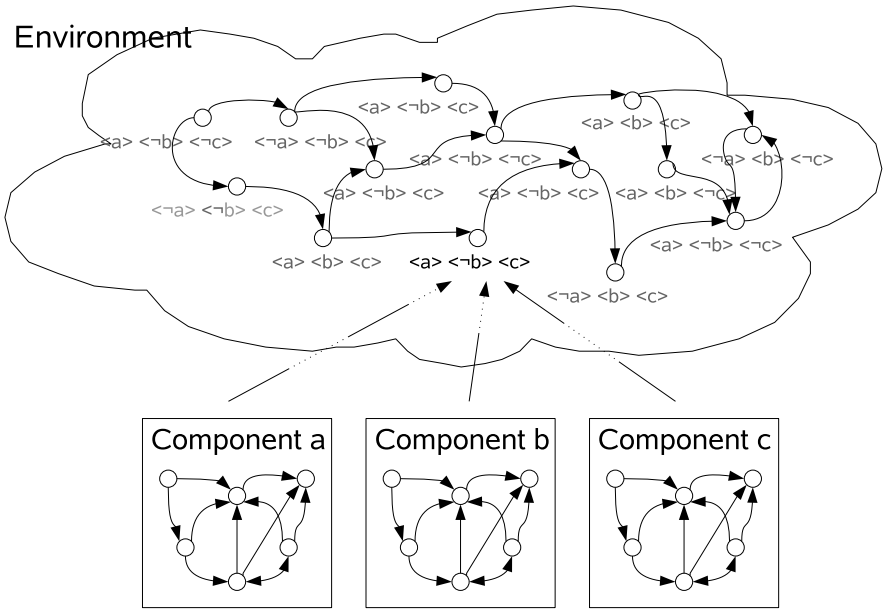

Figure 4.1: Automata Based Model

it has a successful run R_w. In this case an automaton A is not empty.

When constructing such an automaton at the component level, it is called A_c and the following is assumed:

- Initial states, Q_i, are either 'running / waiting' or has not yet entered a state;
- The set of states, Q, corresponds to the states of the component;
- The acceptance condition is defined as reaching one of the following states: terminated, suspended state or fail. These states are in the set Q_f and the acceptance condition is to reach one of these states in Q_f
- The transition conditions are determined by the state change calls of the component.

When the assumed automaton A_c (non-)emptiness procedure establishes that the automaton is not empty, it returns a successful run of A_c. Thus, for any component cycle, when the corresponding automaton has an accepting run, it means that the component's behaviour will eventually hit an accepting state. A simple function $Lab(A_c)$ is defined which returns the following parameters:

- $< a_t >$ - when a component has met the acceptance condition "terminate"
- $< a_s >$ - when a component a has met the acceptance condition "suspended"
- $< a_f >$ - when a component has met the acceptance condition "terminate after going through fail state"
- $< \neg a >$ - when component a has not met any acceptance condition

These labels generated by the function $Lab(A_c)$ will be subsequently collected as a state tree of the environment, the order in which these labels constitute a tree is defined by the order of the same labels passed to it during runtime monitoring.

In the construction of this tree automaton, every state is labelled according to state of components and resources. In this case the transition function is not only related to the state transition of components, but is also tightly bound to the deontic logic accessibility relation. Here it is expected to be able to specify the automaton in the normal form for ECTL$^+$, SNF_{CTL}. Although a rigorous proof of this is not given, it is possible to anticipate that the situation here would be similar to the one in the linear-time case. Namely, in [BCF02], it was shown that a Buchi word automaton can be represented in terms of SNF_{PLTL}, a normal form for PLTL. Similarly, it is reasonable to *expect to be able* to represent a Buchi tree automaton in terms of SNF_{CTL}. Subsequently, this representation of the automaton is enriched by deontic constraints [BB07] and a resolution based verification technique is applied as a verification procedure.

4.3 Formal verification

Different types of deductive verification techniques are available that can deal with the logic SNF_{CTL}. Each of them offer benefits over the others, and two of them have been the focus of this research. Temporal resolution [BB05] has an established search algorithm and has been recently implemented with impressive performance results [BB04]; while Natural deduction, althoug implemented only for linear time settings, has been argued to have the capacity to reduce

complexity [BBG09] in the length of the resolution rather than in its structure. Furthermore, natural deduction has been successfully used in protocol verification [CJM98], in a similar setting to that considered in this research.

4.3.1 Deductive Verification techniques

Among the many proof procedures for temporal logic, the most commonly used in the branching-time setting are Tableau [Wol85, Eme90] and Temporal resolution [BB05]. The choice of one over the other often falls on the development at hand, although in the case of this research, these were two main reasons behind the choice. First, as the developments in branching time logic resolution are quite limited, the choices for formal verification also became limited. Secondly, as the complexity of the specification of a large scale distributed system can grow at a quick pace simply depending on the number of components and resources which constitute the system, it was important to consider developments which have the possibility of reducing the computational requirements, especially when the verification process might have to be performed dynamically at runtime.

In Tableau-based methods, validity of a formula is proven by refutation; the aim of its algorithm is to generate a model from the negated formula's structure: if a model cannot be generated because the structure is empty, then the negated formula is unsatisfiable, and the original formula is therefore valid [Eme90]. This technique, unfortunately, has the downside of not being capable of dealing with the famous induction principle [BD00]. For this research it was concluded that Temporal resolution would be the most beneficial approach to verification, an implementation of which is also available to experiment with [ZHD08]. It was also left open the possibility of utilizing Natural deduction as a verification technique due to its potential in complexity reduction, which could prove essential in applying the procedure to other systems [BBG09].

4.3.2 Temporal resolution for branching time logic

In order to achieve a refutation of a generated specification, two types of resolution rules already defined in [Bol00, BF99] are incorporated: *step* resolution (SRES) and *temporal* resolution (TRES).

Step resolution is used between Formulae that refer to the *same* initial moment of time or *same* next moment along some or all paths. The step resolution rules are given below (where l is a literal and C and D are disjunctions of literals).

SRES 1

$$\frac{\mathbf{start} \Rightarrow C \vee l \qquad \mathbf{start} \Rightarrow D \vee \neg l}{\mathbf{start} \Rightarrow C \vee D}$$

SRES 2

$$\frac{P \Rightarrow \mathbf{A}\bigcirc(C \vee l) \qquad Q \Rightarrow \mathbf{A}\bigcirc(D \vee \neg l)}{(P \wedge Q) \Rightarrow \mathbf{A}\bigcirc(C \vee D)}$$

SRES 3

$$\frac{P \Rightarrow \mathbf{A}\bigcirc(C \vee l) \qquad Q \Rightarrow \mathbf{E}\bigcirc(D \vee \neg l)_{\langle \mathsf{ind} \rangle}}{(P \wedge Q) \Rightarrow \mathbf{E}\bigcirc(C \vee D)_{\langle \mathsf{ind} \rangle}}$$

SRES 4

$$\frac{P \Rightarrow \mathbf{E}\bigcirc(C \vee l)_{\langle \mathsf{ind} \rangle} \qquad Q \Rightarrow \mathbf{E}\bigcirc(D \vee \neg l)_{\langle \mathsf{ind} \rangle}}{(P \wedge Q) \Rightarrow \mathbf{E}\bigcirc(C \vee D)_{\langle \mathsf{ind} \rangle}}$$

Table 4.7: Step resolution rules

When an empty constraint is generated on the right hand side of the conclusion of the resolution rule, a constant **false** is introduced to indicate this terminating clause.

Now let us present the temporal resolution rules; in the formulation of the rules below l is a literal and the first premises abbreviate the **A** and **E** loops in l [BD00], i.e. the situation where, given that P is satisfied at some point of time, l occurs always from that point on all or some path respectively.

TRES 1

$$\frac{P \Rightarrow \mathbf{A}\bigcirc\mathbf{A}\,\Box l \qquad Q \Rightarrow \mathbf{A}\diamondsuit\neg l}{Q \Rightarrow \mathbf{A}(\neg P\,\mathcal{W}\,\neg l)}$$

TRES 2

$$\frac{P \Rightarrow \mathbf{A}\bigcirc\mathbf{A}\,\Box l \qquad Q \Rightarrow \mathbf{E}\diamondsuit\neg l_{\langle LC(\mathsf{ind}) \rangle}}{Q \Rightarrow \mathbf{E}(\neg P\,\mathcal{W}\,\neg l)_{\langle LC(\mathsf{ind}) \rangle}}$$

TRES 3

$$\frac{P \Rightarrow \mathbf{E}\bigcirc\mathbf{E}\,\Box l_{\langle LC(\mathsf{ind}) \rangle} \qquad Q \Rightarrow \mathbf{A}\diamondsuit\neg l}{Q \Rightarrow \mathbf{A}(\neg P\,\mathcal{W}\,\neg l)}$$

TRES 4

$$\frac{P \Rightarrow \mathbf{E}\bigcirc\mathbf{E}\,\Box l_{\langle LC(\mathsf{ind}) \rangle} \qquad Q \Rightarrow \mathbf{E}\diamondsuit\neg l_{\langle LC(\mathsf{ind}) \rangle}}{Q \Rightarrow \mathbf{E}(\neg P\,\mathcal{W}\,\neg l)_{\langle LC(\mathsf{ind}) \rangle}}$$

Table 4.8: Temporal resolution rules

Example Verification - $\mathrm{SNF}_{\mathrm{CTL}}$

It is now considered a sample property that should be verified in the formula

$$\ddagger \qquad \mathbf{A}(\,\Box\diamondsuit p \wedge \diamondsuit\,\Box\neg p)$$

Below it is shown how this formula can be represented in terms of $\mathrm{SNF}_{\mathrm{CTL}}$ and then applied to this specification the resolution technique as a verification method, using the resolution rules detailed in the previous section.

To verify ($\ddagger$) the resolution method is applied to the set of $\mathrm{SNF}_{\mathrm{CTL}}$ clauses $\mathrm{SNF}_{\mathrm{CTL}}(\ddagger)$. The resolution proof commences by presenting at steps 1 – 13 the clauses of $\mathrm{SNF}_{\mathrm{CTL}}(\ddagger)$ in the following order: initial clauses, step clauses and, finally, any sometime clauses.

1. $\mathbf{start} \Rightarrow x$
2. $\mathbf{start} \Rightarrow \neg x \vee y$
3. $\mathbf{start} \Rightarrow \neg x \vee x_1$
4. $\mathbf{start} \Rightarrow \neg z \vee \neg p$
5. $\mathbf{start} \Rightarrow \neg z \vee z_1$
6. $\mathbf{true} \Rightarrow \mathbf{A}\bigcirc(\neg z \vee \neg p)$
7. $\mathbf{true} \Rightarrow \mathbf{A}\bigcirc(\neg z \vee z_1)$
8. $x_1 \Rightarrow \mathbf{A}\bigcirc y$
9. $x_1 \Rightarrow \mathbf{A}\bigcirc x_1$
10. $z_1 \Rightarrow \mathbf{E}\bigcirc\neg p_{\langle \mathsf{f}\rangle}$
11. $z_1 \Rightarrow \mathbf{E}\bigcirc z_{1\langle \mathsf{f}\rangle}$
12. $y \Rightarrow \mathbf{A}\diamondsuit p$
13. $x \Rightarrow \mathbf{E}\diamondsuit z_{\langle LC(\mathsf{f})\rangle}$

Step resolution rules are applied between 1 and 2, and 1 and 3. No more SRES rules are applicable. Formula 12 is an eventuality clause, and therefore, we are looking for a loop in $\neg p$ (see [BD00] for the formulation of the loop searching procedure). The desired loop, $\mathbf{E}\,\Box\mathbf{E}\bigcirc\neg p_{\langle LC(\mathsf{f})\rangle}$ (given that condition z_1 is satisfied) can be found considering clauses 10 and 11. Thus, apply the TRES 3 rule to resolve this loop and clause 12, obtaining 16. Next it is removed $\mathbf{E}\,\mathcal{W}$ from 16 deriving a purely classical Formula 17 (y is a new variable). Simplify the latter, apply TEMP (the 'temporising' rule, see [Bol00], obtaining, in particular, 19 and 20, and then a series of SRES rules to newly generated clauses. Now, as no more SRES rules are applicable, we find another eventuality, Formula 13, and thus proceed to look for a loop in

$\neg z$. This loop can be found considering Formulae 9 and 26: $\mathbf{A}\bigcirc\mathbf{A}\,\Box\neg z$ given that condition x_1 is satisfied. Thus, apply TRES 2 to resolve this loop and 13 deriving 27. Then remove $\mathbf{E}\,\mathcal{W}$ from the latter (on step 28, where w is a new variable, used only one of its conclusions). Applying simplification and temporising to 28 it is obtained 29. The desired terminating clause **start** $\Rightarrow$ **false** is deduced by applying SRES 1 to steps 1, 15 and 23.

14.	**start**	$\Rightarrow$	y	$1, 2,\ SRES\ 1$
15.	**start**	$\Rightarrow$	x_1	$1, 3,\ SRES\ 1$
16.	y	$\Rightarrow$	$\mathbf{A}(\neg z_1\,\mathcal{W}\,p)$	$10, 11, 12\ TRES\ 3$
17.	y	$\Rightarrow$	$p \vee \neg z_1 \wedge v$	16, $\mathbf{A}\,\mathcal{W}$ Removal
18.	v	$\Rightarrow$	$\mathbf{A}\bigcirc(p \vee \neg z_1 \wedge v)$	16, $\mathbf{A}\,\mathcal{W}$ Removal
19.	**start**	$\Rightarrow$	$\neg y \vee p \vee \neg z_1$	17, SIMP, TEMP
20.	**true**	$\Rightarrow$	$\mathbf{A}\bigcirc(\neg y \vee p \vee \neg z_1)$	17, SIMP, TEMP
21.	**start**	$\Rightarrow$	$p \vee \neg z_1$	$14, 19,\ SRES\ 1$
22.	**start**	$\Rightarrow$	$p \vee \neg z$	$5, 21,\ SRES\ 1$
23.	**start**	$\Rightarrow$	$\neg z$	$4, 22,\ SRES\ 1$
24.	x_1	$\Rightarrow$	$\mathbf{A}\bigcirc(p \vee \neg z_1)$	$8, 20,\ SRES\ 3$
25.	x_1	$\Rightarrow$	$\mathbf{A}\bigcirc(p \vee \neg z)$	$7, 24,\ SRES\ 3$
26.	x_1	$\Rightarrow$	$\mathbf{A}\bigcirc\neg z$	$6, 25,\ SRES\ 3$
27.	x	$\Rightarrow$	$\mathbf{E}(\neg x_1\,\mathcal{W}\,z)_{\langle LC(\mathsf{f})\rangle}$	$9, 26, 13\ TRES\ 2$
28.	x	$\Rightarrow$	$z \vee \neg x_1 \wedge w$	27 $\mathbf{E}\,\mathcal{W}$ Removal
29.	**start**	$\Rightarrow$	$\neg x \vee z \vee \neg x_1$	28 SIMP, TEMP
30.	**start**	$\Rightarrow$	**false**	$1, 15, 23\ SRES\ 1$

A contradiction is found, meaning that $\mathrm{SNF}_{\mathrm{CTL}}(\ddagger)$ itself is unsatisfiable; in fact, correctness of the transformation of ECTL$^+$ formulae into $\mathrm{SNF}_{\mathrm{CTL}}$ [BB06], as well as termination and correctness of the resolution method defined over $\mathrm{SNF}_{\mathrm{CTL}}$ [Bol00, BF99], allows to utilize the latter as refutation procedure for ECTL$^+$.

A sample of the $\mathrm{SNF}_{\mathrm{CTL}}$ state tree is shown in figure 4.2, here it is possible to identify where the proof looks for a loop in $\neg p$ (bottom branch), and a loop in $\neg z$ (top branch), and the contradiction of z and $\neg z$ led by the eventuality in clause 13.

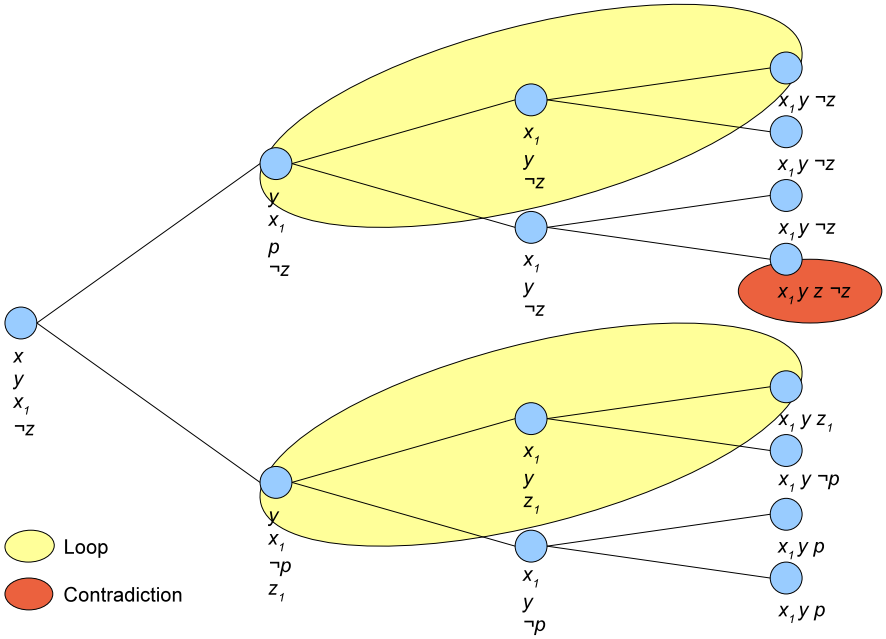

Figure 4.2: States Tree Example

Example Verification - $\mathrm{SNF}^{D}_{\mathrm{CTL}}$

Let us now analyze more in detail the core concept of Temporal Deontic Specification (TDS). It is supposed that the given specification is either directly written in terms of TDS or in the language of ECTL^{+}_{D}. In the latter case the formulae of the specification must be transformed into the desired form of TDS. Since ECTL^{+}_{D} extends ECTL^{+} by allowing deontic constraints and similarly since TDS is a deontic extension of $\mathrm{SNF}_{\mathrm{CTL}}$, a normal form for the logic ECTL^{+}, [BB06] it is possible to simply enrich the transformation procedure of [BB06] by the corresponding rules dealing with the deontic operations. Indeed, due to the fact that there is no interaction between temporal and deontic constraints, the only rule that is needed for the transformation of the formulae with deontic constraints is the renaming rule, which would work in a similar way to [DFB02], i.e. which would allow renaming of an embedded deontic subformula by a new

proposition.

Renaming

Given $P \Rightarrow Q(F)$ it is possible to derive $P \Rightarrow Q(F/x)$ and $x \Rightarrow F$, where $Q(F)$ is a formula with the designated subformula F and $Q(F/x)$ means a result of replacing F by a new proposition symbol x in Q.

Temporising

Given a purely classical expression $A \Rightarrow B$ it is possible to transform it into $\mathbf{start} \Rightarrow \neg A \vee B$ and $\mathbf{true} \Rightarrow \mathbf{A}\bigcirc(\neg A \vee B)$. In particular, the following case of this rule can be applied: from $\mathbf{true} \Rightarrow A \vee B$ derive $\mathbf{start} \Rightarrow \neg A \vee B$ and $\mathbf{true} \Rightarrow \mathbf{A}\bigcirc(\neg A \vee B)$.

Removal of $\mathbf{A}\,\mathcal{W}$

Recall that the formula of the type $\mathbf{A}(A\,\mathcal{W}\,B)$ or $\mathbf{E}(A\,\mathcal{W}\,B)$ can appear as a result of the application of Temporal Resolution rules. Hence, it is necessary to transform a resolvent of this type into the desired form. This is first achieved by the application of the $\mathcal{W}$ removal rule. In particular, by applying the $\mathbf{A}\,\mathcal{W}$ removal rule: given $P \Rightarrow \mathbf{A}(A\,\mathcal{W}\,B)$ derive $P \Rightarrow B \vee (A \wedge x)$ and $x \Rightarrow \mathbf{A}\bigcirc(B \vee (A \wedge x))$ [Bol00].

Example. Now, consider an example specification in which essentially a normative framework is used for reconfiguration, and where a model is requested to be updated.

Let r and s represent two components that can be bound to the system. Further let q be a new composite component, a composition of r and s. Next, let r and s be such that r always requires its counterpart component, s, not to be active in any of the next states and s requires r not to be bound (i.e. enabled) in some possible development of the system, i.e. at the successor state in the direction $\langle f \rangle$. Additionally, let us assume that the specification of the system requires that this new component, q, should not be bound at the next state. This is represented by the following formula of ECTL^{+}_{D}:

$$(\dagger) \qquad \mathbf{A}\,\Box(r \Rightarrow (\mathbf{A}\bigcirc s \wedge \mathcal{O}_i\neg q)) \wedge \mathbf{A}\,\Box(s \Rightarrow (\mathbf{E}\bigcirc r \wedge \mathcal{O}_i\neg q))$$

Finally, let the system receive a request for the permission to eventually bind q whichever way

it evolves:

$$(\ddagger) \qquad \mathbf{start} \Rightarrow \mathbf{A}\diamondsuit\mathcal{P}_i q$$

The specification technique requires to transform formulae (†) and (‡) to the structure required by TDS. This translation, as mentioned above, when it concerns the temporal part, is described in the previous work [BBB+06, BB06] and if it involves deontic constraints then additionally it uses standard classical transformations towards normal forms and the renaming rule. To simplify the reading of the thesis, the rules involved in the examples are presented below.

In the table below the conditions of the component system and their representations in the language of TDS (note that w is a new (auxiliary) proposition introduced to achieve the required form of TDS clauses) are summarized. Recall that each clause of TDS is in the scope of the $\mathbf{A}\,\square$ and this common prefix for the TDS clauses is omitted in the rest of the thesis to simplify reading. Also, recall that each **E** step clause would have to be labelled by a specific label f while every **E** sometime clause by some index $LC(ind)$ [Bol00].

Conditions of the System	Constraints of TDS
Dependency between counterpart components	$r \Rightarrow \mathbf{A}\bigcirc s$ $\mathbf{true} \Rightarrow \neg r \vee \mathcal{O}_i \neg q$ $s \Rightarrow \mathbf{E}\bigcirc r$ $\mathbf{true} \Rightarrow \neg s \vee \mathcal{O}_i \neg q$
A request for the permission to eventually bind q	$\mathbf{start} \Rightarrow x$ $x \Rightarrow \mathbf{A}\diamondsuit w$ $\mathbf{true} \Rightarrow \neg w \vee \mathcal{P}_i q$

Table 4.9: Conditions of the system and Constraints of TDS

The procedure begins by updating the set of resolution rules developed for $\mathrm{SNF}_{\mathrm{CTL}}$ [BF99] by new resolution rules capturing the deontic constraints. Recall that among the set of the TDS clauses are initial clauses, step clauses, eventuality formulae, and deontic clauses. In order to achieve refutation three types of resolution rules are applied: Step Resolution (classical resolution) (SRES), Temporal Resolution (TRES) - described in the previous section, and deontic resolution (DRES).

When TDS clauses contain eventualities then the resolution procedure tackles the cases where such promises for the events to occur contradict some invariants, or loops [Bol00]. It is only noted here that loops are formulae that constrain some proposition to be always true (on all or some paths) given some conditions hold. Finally, when two deontic clauses contain complementary constraints, $O_i l$ and $P_i \neg l$ then it is allowed to apply the new, deontic resolution rule, which, in fact, works similarly to the modal resolution rule in [DFB02].

$$\frac{\begin{array}{l}DRES\\ \textbf{true} \Rightarrow D \vee O_i l\\ \textbf{true} \Rightarrow D' \vee P_i \neg l\end{array}}{\textbf{true} \Rightarrow D \vee D'}$$

Table 4.10: Deontic resolution rule

Here is presented a resolution based refutation for the set of clauses of TDS obtained for the component system analysed in the previous section.

TDS

1.	r	$\Rightarrow$	$\mathbf{A}\bigcirc s$	
2.	**true**	$\Rightarrow$	$\neg r \vee \mathcal{O}_i \neg q$	
3.	s	$\Rightarrow$	$\mathbf{E}\bigcirc r_{\langle f \rangle}$	
4.	**true**	$\Rightarrow$	$\neg s \vee \mathcal{O}_i \neg q$	
5.	**start**	$\Rightarrow$	x	
6.	x	$\Rightarrow$	$\mathbf{A}\diamondsuit w$	
7.	**true**	$\Rightarrow$	$\neg w \vee \mathcal{P}_i q$	

$Proof$

8.	**true**	$\Rightarrow$	$\neg s \vee \neg w$	$DRES$ 4, 7
9.	r	$\Rightarrow$	$\mathbf{A}\bigcirc \neg w$	$SRES$ 2, 1, 8
10.	s	$\Rightarrow$	$\mathbf{A}\bigcirc \neg w$	$from$ 8
11.	$r \vee s$	$\Rightarrow$	$\mathbf{E}\bigcirc \Box \neg w_{\langle f \rangle}$	1, 3, 9, 10
12.	x	$\Rightarrow$	$\neg(r \vee s)\, \mathcal{W}\, w$	$TRES$, 6, 11
13.	x	$\Rightarrow$	$w \vee \neg(r \vee s)$	$\mathcal{W}$ $removal$, 12
14.	x	$\Rightarrow$	$w \vee \neg r$	$classical$, 13
15.	x	$\Rightarrow$	$w \vee \neg s$	$classical$, 13
16.	**start**	$\Rightarrow$	$\neg x \vee w \vee \neg r$	$temporising$, 14
17.	**start**	$\Rightarrow$	$\neg x \vee w \vee \neg s$	$temporising$, 15
18.	**start**	$\Rightarrow$	$\neg s \vee \neg w$	$temporising$, 8
19.	**start**	$\Rightarrow$	$\neg s$	$SRES$1, 5, 17, 18

In this proof, step 8 is obtained by the application of deontic resolution to 4 and 7. Step 9 is the application of Step Resolution to 1 and 8 (recall that from **true** $\Rightarrow \neg s \vee \neg w$ by temporising it is possible to derive **start** $\Rightarrow \neg s \vee \neg w$ and **true** $\Rightarrow \mathbf{A}\bigcirc(\neg s \vee \neg w)$, and use the latter to resolve with 1). Thus, from **true** $\Rightarrow \mathbf{A}\bigcirc(\neg s \vee \neg w)$ we also have step 10.

Now, since there is an eventuality clause 6, $x \Rightarrow \mathbf{A}\Diamond w$, the resolution based verification technique searches for a loop in $\neg w$. The desired loop can be found combining together clauses 1, 3, 9 and 10 to give us: $r \vee s \Rightarrow \mathbf{E}\bigcirc \Box\neg w_{\langle f \rangle}$. This loop being resolved with the eventuality clause, produces the resolvent on step 12. Remove $\mathcal{W}$ from this resolvent, deducing 13. Subsequent classical transformation and the temporising rule guide the deduction of steps 14-18. Finally, applying Step Resolution, step 19 is derived.

As mentioned, it is possible to describe how our system can identify normative invariants which should be preserved, and also detect hidden invariants, i.e. those that are not explicitly given in the specification.

Analysing the proof above it is known that s should not be initially active. Note that the procedure has detected a loop (invariant) in $\neg w$ which is immediately obvious from the set of TDS clauses. Additionally, this loop, in conjunction with clauses 2 and 7 indicates a hidden 'deontic invariant' property, that s fires the condition $\mathcal{O}_i\neg q$ and w fires the condition $\mathcal{P}_i q$. Now, if it is assumed that r is initially active, then it is possible to continue the proof above and derive a contradiction as follows:

20.	**start**	$\Rightarrow$	r	*assumption*
21.	**start**	$\Rightarrow$	w	$SRES$ 1, 1, 16, 20
22.	**start**	$\Rightarrow$	$\mathcal{P}_i q$	$SRES$1, 7, 21
23.	**start**	$\Rightarrow$	$\mathcal{O}_i\neg q$	$SRES$1, 2, 20
24.	**start**	$\Rightarrow$	**false**	$DRES$, 22, 23

Thus, a request to bring a composite component, q to the system can only be satisfied if r is not active. Otherwise, if r is bound to the system, the request to bind q should be rejected.

In the deontic language, it is possible to set up some predefined accessibilities (on the top of those that are defined for every model - such as transitivity, etc) which are called in this research 'deontically accessible worlds'. During the reconfiguration it is intended to arrive at a deontically accessible world to update the model, it is possible to do this in two ways:

1. When such a world that corresponds to the reconfiguration specification can be found in the model and it is deontically accessible;

2. When we cannot find such a world we want to update the model, this update should then satisfy both, the criteria of reconfiguration and this deontic accessibility.

4.3.3 Natural deduction

Natural deduction is an approach to deductive verification that aims at providing a deductive proof system that would be "as close as possible to actual reasoning" [Gen35], in contrast with a proof through axioms. The importance of natural deduction procedures has grown over time, mainly due to its ability to emphasize the goal-directed nature of proofs, which can give a more human understandable insight into the proof results.

The computer science community has become more interested in Natural Deduction systems [BMV96, Pfe01] for being applicable in many AI areas, most notably, in agent engineering [Woo00]. This, together with the recent interest in normative systems (see for example, DEON workshop series [LW06]), makes the development of a natural deduction technique an important task. Among other interesting and even surprising applications of ND systems is for example their use in the verification of security protocols [CJM98].

Here is presented the formulation of CTL^D with a slightly different set of rules for its temporal part in comparison with its original formulation in [BGS06]. In particular, it is introduced a new rule for the $\mathcal{U}$ operation paired with a path quantifier, which reduces the overall number of the rules in the temporal part of the system (see details below).

Extended CTL^D Syntax and Semantics

To define the rules of the natural system the syntax of CTL^D is extended by introducing labelled formulae.

Firstly, it is defined the set of labels, terms of the set $Lab = Lab_S \cup Lab_P$ where $Lab_S = \{x, y, z, \ldots\}$ is a set of variables interpreted over states of a tree and $Lab_P = \{\alpha, \beta, \gamma, \ldots\}$ is a set of variables over paths of a tree.

Universal and rigid variables are distinguished. This second type of variables is linked with the restrictions on the application of some of the rules which will be explained later. In the rest of the chapter it is referred to the sets of labels that represent universal and rigid variables as to Lab_S^{univ}, Lab_P^{univ} and Lab_S^{rigid}, Lab_P^{rigid}, respectively.

The equality over the labels, $\simeq$ is needed, and the generalisation of the operations '$\prec$' and '$\preceq$' introduced in the linear-time case [BBGS06, BGS07] such that $(i \prec j)_\varphi$ and $(i \preceq j)_\varphi$ abbreviate, respectively, that $i \prec j$ and $i \preceq j$, hold in an (arbitrary or specific) branch φ (depending on whether φ is universal or rigid), which starts at state i and includes j. In other words, it is assumed that the starting point of path φ is the state that corresponds to the first state variable, i, in the relations $\prec$ and $\preceq$. The following properties for these relations hold. For any $i, j, k \in Lab_S$ and $\varphi, \psi \in Lab_P$

- $(i \preceq i)_\varphi$ (reflexivity),
- if $(i \preceq j)_\varphi$ and $(j \preceq k)_\psi$ then $(i \preceq k)_\chi$ (transitivity), where
  - χ is a new label from Lab_P^{rigid} if one of the φ or ψ, or both of them, are from Lab_P^{rigid},
  - otherwise, $\chi \in Lab_P^{univ}$;

Now, it is defined the relation $Next$ corresponding to the 'predecessor-successor' relation in a tree, again, generalising it from the linear-time case as follows: $Next(i, j)_\varphi$ means that j is an immediate successor of i on an arbitrary (if $\varphi \in Lab_P^{univ}$) or specific (if $\varphi \in Lab_P^{rigid}$) path. $Next$ satisfies the seriality property requiring that any state has a successor state.

Let $'$ abbreviate the operation which being applied to a state label i gives us the state label i' such that $Next(i, i')_\varphi$.

Finally, it is introduced the deontic accessibility relation, D, over the state labels as follows:

- for all $i \in Lab_S$ there exists $j \in Lab_S$ such that $D_a(i, j)$, for all $a \in Ag$ (seriality of D_a);
- if $D_a(i, j)$ and $D_a(j, k)$, then $D_a(i, k)$ (transitivity of D_a) for all $a \in Ag$;
- if $D_a(i, j)$ and $D_a(i, k)$, then $D_a(j, k)$ (Euclidian).

As previously, following [Sim94], the expressions representing the properties of relations above are called *relational judgements*.

Definition 12 ($\mathbf{CTL^D_{ND}}$ Syntax) *If A is a CTL^D formula and $i \in Lab_S$ then $i : A$ is a $\mathrm{CTL}^D_{\mathrm{ND}}$ formula. Any relational judgement of the type $Next(i, i')_\varphi$, $Next(i, i')_{sf\varphi_k}$, $(i \preceq j)_\varphi$, and $(i \preceq j)_{sf\varphi}$, $D_a(i, j)$ where $i, j \in Lab_S$, $\varphi \in Lab_P$, and $a \in Ag$ is a $\mathrm{CTL}^D_{\mathrm{ND}}$ formula.*

Some useful and rather straightforward properties relating operations on labels are given below.

- if $(i \prec j)_\varphi$ then $(i \preceq j)_\varphi$,
- if $Next(i, j)_\varphi$ then $(i \preceq j)_\varphi$.

It is easy to see, looking at the properties of all these relational judgements introduced above, that they correspond to the properties of the deontic extension of CTL models.

$\mathbf{CTL^D_{ND}}$ Semantics. For the interpretation of $\mathrm{CTL}^D_{\mathrm{ND}}$ formulae the semantic constructions previously defined for the logic $\mathrm{CTL}^D$are adapted. From now on, the capital letters $A, B, C, D, \ldots$ are used as metasymbols for CTL^D formulae, and calligraphic letters $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D} \ldots$ to abbreviate formulae of $\mathrm{CTL}^D_{\mathrm{ND}}$, i.e. either labelled formulae or relational judgements. The intuitive meaning of $i{:}A$ is that A is satisfied in the world $i \in Lab_S$. Let Γ be a non-empty set of $\mathrm{CTL}^D_{\mathrm{ND}}$ formulae, let $Lab^\Gamma_S = \{x \mid x{:}A \in \Gamma\}$, $Lab^\Gamma_P = \{\varphi \mid \mathcal{B}_\varphi \in \Gamma\}$ (where $\mathcal{B}$ abbreviates

relational judgements), let $\mathcal{M} = \langle S, R, s_0, X, L\rangle$ be a model, and let χ be a set of paths. For the purpose of constructing semantics for $\mathrm{CTL}^{D}_{\mathrm{ND}}$ we shall use very natural functions $g^{\mathcal{M}}_{S}$ and $g^{\mathcal{M}}_{P}$ from Lab^{Γ}_{S} to S and from Lab^{Γ}_{P} to χ respectively. Now it is possible to introduce direct analogs of the notions of satisfiability and validity for the extended semantics. For example, formula $i : A$ is satisfiable in extended semantics if it is satisfiable in the usual sense in some model $\mathcal{M}$ at some point $g^{\mathcal{M}}_{S}(i)$.

$\mathbf{CTL^{D}_{ND}}$ Rules

Below it is defined the sets of elimination and introduction rules for Booleans, where '*el*' and '*in*' that follow a Boolean operation abbreviate an elimination or an introduction rule for this operation.

Elimination Rules :		Introduction Rules :	
$\wedge\ el_1$	$\dfrac{i:A \wedge B}{i:A}$ $\wedge\ el_2$ $\dfrac{i:A \wedge B}{i:B}$	$\wedge\ in$	$\dfrac{i:A \quad i:B}{i:A \wedge B}$
$\vee\ el$	$\dfrac{i:A \vee B \quad i:\neg A}{i:B}$	$\vee\ in_1$	$\dfrac{i:A}{i:A \vee B} \vee\ in_2 \dfrac{i:B}{i:A \vee B}$
$\Rightarrow el$	$\dfrac{i:A \Rightarrow B \quad i:A}{i:B}$	$\Rightarrow\ in$	$\dfrac{[i:C] \quad i:B}{i:C \Rightarrow B}$
$\neg\ el$	$\dfrac{i:\neg\neg A}{i:A}$	$\neg\ in$	$\dfrac{[j:C] \quad i:B \quad i:\neg B}{j:\neg C}$

Table 4.11: $\mathrm{CTL}^{D}_{\mathrm{ND}}$ rules for Boolean

In the formulation of the rules '$\Rightarrow$ *in*' and '$\neg$ *in*' formulae $[i:C]$ and $[j:C]$ respectively must be the most recent non-discarded [BBGS05] assumptions occurring in the proof. When it is applied one of these rules on step n and discharge an assumption on step m, all formulae from m to $n-1$ are discarded. It is possible to write $[m-(n-1)]$ to indicate this situation.

Next, it is introduced the sets of rules for the temporal part.

Elimination Rules

$$\mathbf{A}\bigcirc el \quad \frac{i{:}\mathbf{A}\bigcirc A}{Next(i,i')_{\varphi},\quad i' : A} \qquad \mathbf{E}\bigcirc el \quad \frac{i{:}\mathbf{E}\bigcirc A}{Next(i,i')_{\varphi_r},\quad i' : A}$$
$$i' : A \in M1 \qquad i' : A \in M1$$

$$\mathbf{A}\,\Box_{el} \quad \frac{i{:}\mathbf{A}\,\Box A}{(i \preceq j)_{\varphi},\quad j{:}A} \qquad \mathbf{E}\,\Box_{el} \quad \frac{i{:}\mathbf{E}\,\Box p}{(i \preceq j)_{\varphi_r},\quad j{:}A}$$

$$\mathbf{E}\Diamond_{el} \quad \frac{i{:}\mathbf{E}\Diamond A}{(i \preceq j_r)_{\varphi_r},\quad j{:}A} \qquad \mathbf{A}\Diamond_{el} \quad \frac{i{:}\mathbf{A}\Diamond A}{(i \preceq j_r)_{\varphi},\quad j{:}A}$$
$$j \mapsto i,\ \forall C(j{:}C \notin M1) \qquad j \mapsto i,\ \forall C(j{:}C \notin M1)$$

$$\mathcal{O}_{el} \quad \frac{i{:}\mathcal{O}_a A}{D_a(i,j),\ j{:}A} \qquad \mathcal{P}_{el} \quad \frac{i{:}\mathcal{P}_a A}{D_a(i,j_r),\ j{:}A\ j \mapsto i,\ \forall C(j{:}C \notin M_1)}$$

$$\mathbf{P}\,\mathcal{U}_{\,el} \quad \frac{i{:}\mathbf{P}(A\,\mathcal{U}\,B),\ i{:}\mathbf{A}\,\Box(B \Rightarrow C),\quad i{:}\mathbf{A}\,\Box(A \wedge \mathbf{P}\bigcirc C) \Rightarrow C}{i{:}C}$$

Table 4.12: $\mathrm{CTL}^{D}_{\mathrm{ND}}$ elimination rules for temporal and deontic operations

- If a type of a variable in a premise of a rule is not indicated then it can be either universal or rigid. A variable which is not indicated as rigid is universal.

- In $\mathbf{A}\bigcirc_{el}$ and $\mathbf{E}\bigcirc_{el}$ rules the conclusion $i' : A$ is marked by M_1. The condition $\forall C(j : C \notin M1)$ means that the label j should not occur in the proof in any formula, C, that is marked by $M1$ while the condition $j{:}A \notin M1$ means that $j{:}A$ is not marked by $M1$.

- In $\mathbf{A}\,\Box_{in}$, $\mathbf{E}\,\Box_{in}$ and $\mathcal{O}_{in}$, the abbreviation '[]' for a relational judgement in their premises mean that if $(i \preceq j)_{\varphi}$ or $D_a(i,j)$ is an assumption, then it must be the most recent assumption that must be discarded. Applying the rule on step n of the proof, it is discarded $(i \preceq j)_{\varphi}$ and all formulae until the step n.

- When it is applied a rule where rigid variables are introduced in its conclusion, it is picked a new variable from a list of available rigid variables. A newly introduced rigid world variable relatively binds the other variable in the relational judgement; it is similar to PLTL - this binding relation is transitive but cannot be reflexive.

Below the rules for relational judgements are introduced.

$\mathbf{E}\bigcirc_{in}$ $\dfrac{i' : A \quad Next(i, i')_\varphi}{i : \mathbf{E}\bigcirc A}$

$\mathbf{A}\bigcirc_{in}$ $\dfrac{i' : A \quad Next(i, i')_\varphi}{i : \mathbf{A}\bigcirc A}$ $\varphi \notin Lab_P^{rigid},\ i' \notin Lab_S^{rigid}$

$\mathbf{A}\,\Box_{in}$ $\dfrac{j : A, \quad [(i \preceq j)_\varphi]^\star}{i : \mathbf{A}\,\Box A}$ $\varphi \notin Lab_p^{rigid},\ j \notin Lab_s^{rigid},\ j : A \notin M1$

$\mathbf{E}\,\Box_{in}$ $\dfrac{j : A, \quad [(i \preceq j)_\varphi]^\star}{i : \mathbf{E}\,\Box A}$ $j \notin Lab_s^{rigid},\ j : A \notin M1$

$\mathbf{A}\Diamond_{in}$ $\dfrac{j : A, \quad (i \preceq j)_\varphi}{i : \mathbf{A}\Diamond A}$ $\varphi \notin Lab_p^{rigid}$

$\mathbf{E}\Diamond_{in}$ $\dfrac{j : A, \quad (i \preceq j)_\varphi}{i : \mathbf{E}\Diamond A}$

$\mathbf{E}\,\mathcal{U}_{in}$ $\dfrac{i : B \vee (A \wedge \mathbf{E}\bigcirc\mathbf{E}(A\,\mathcal{U}\,B))}{i : \mathbf{E}(A\,\mathcal{U}\,B)}$

$\mathbf{A}\,\mathcal{U}_{in}$ $\dfrac{i : B \vee (A \wedge \mathbf{A}\bigcirc\mathbf{A}(A\,\mathcal{U}\,B))}{i : \mathbf{A}(A\,\mathcal{U}\,B)}$

$\mathcal{O}_{in}$ $\dfrac{j : A,\ [D_a(i, j)]^\star}{i : \mathcal{O}_a A}$ $j \notin Lab_s^{rigid},\ j : A \notin M1$

$\mathcal{P}_{in}$ $\dfrac{j : A,\ D_a(i, j)}{i : \mathcal{P}_a A}$

Table 4.13: $\mathrm{CTL}^D_{\mathrm{ND}}$ introduction rules for temporal and deontic operations

$\preceq$ *reflexivity* $\dfrac{}{(i \preceq i)_\chi}$ *where* $i \in M1$

$\preceq$ *transitivity* $\dfrac{(i \preceq j)_\chi,\ (j \preceq k)_\varphi}{(i \preceq k)_\psi}$

$\bigcirc$ *seriality* $\dfrac{}{Next(i, i')_\chi}$

$\bigcirc / \preceq$ $\dfrac{Next(i, i')_\chi}{(i \preceq i')_\chi}$

$\prec / \preceq$ $\dfrac{(i \prec j)_\chi}{(i \preceq j)_\chi}$

D seriality $\dfrac{}{D_a(i, k)_\chi}$

D transitivity $\dfrac{D_a(i, j),\ D_a(j, k)}{D_a(i, l)}$

D Euclidian $\dfrac{D(i, j), D_a(i, k)}{D_a(j, k)}$

Table 4.14: $\mathrm{CTL}^D_{\mathrm{ND}}$ rules for relational judgements

The $\preceq$ transitivity rule requires that $\psi \in Lab_P^{rigid}$ is a new label, if at least one of χ or φ are elements of Lab_P^{rigid}, and $\psi \in Lab_P^{univ}$ otherwise.

Definition 13 ($\mathrm{CTL}^D_{\mathrm{ND}}$ **proof)** *An ND proof of a* CTL^D *formula* B *is a finite sequence of* $\mathrm{CTL}^D_{\mathrm{ND}}$ *formulae* $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ *which satisfies the following conditions: every* $\mathcal{A}_i$ $(1 \leq i \leq n)$ *is either an assumption, in which case it should have been discarded, or the conclusion of one of the ND rules, applied to some foregoing formulae, the last formula,* $\mathcal{A}_n$*, is* $x : B$*, for some label* x*, no rigid variable – world or path label – occurs in the conclusion or relatively binds itself.*

Correctness

The following two theorems characterise the correctness argument.

Theorem 1 *[Soundness of* $\mathrm{CTL}^{D}_{\mathrm{ND}}$*]*

Let $\mathfrak{D} = \langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k \rangle$ *be a proof of* $\mathrm{CTL}^{D}_{\mathrm{ND}}$ *formula* $\mathcal{B}$*. Then* $\models_{ND} B$*.*

Theorem 2 *[*$\mathrm{CTL}^{D}_{\mathrm{ND}}$ *Completeness] For any* CTL^{D} *formula, A, if* $\models_{ND} A$ *then there exists a* $\mathrm{CTL}^{D}_{\mathrm{ND}}$ *proof of A.*

Proofs of these theorems can be found in the technical report available from [BG08].

Natural Deduction Example

Here it is given an example of some proofs including the famous induction principle in CTL. Note that this proof in turn uses some other derived rules and it is again referred the reader to the technical report mentioned above for the full account of details. Note that in the proofs below it is allowed to introduce theorems as steps of these proofs and some derived rules.

$$\vdash \neg \mathbf{E} \Diamond A \Rightarrow \mathbf{A} \Box \neg A \tag{4.1}$$

1. $x : \neg \mathbf{E} \Diamond A$	*assumption*
2. $(x \preceq y)_{\alpha}$	*assumption*
3. $y : A$	*assumption*
4. $x : \mathbf{E} \Diamond A$	$2, 3, \mathbf{E} \Diamond_{in}$
5. $y : \neg A$	$1, 4, \neg_{in}, [3-4]$,
6. $x : \mathbf{A} \Box \neg A$	$5, \mathbf{A} \Box_{in}, [2-5], \alpha \notin Lab^{P}_{Rigid},\ y \notin Lab^{S}_{Rigid}$
7. $x : \neg \mathbf{E} \Diamond A \Rightarrow \mathbf{A} \Box A$	$6, \Rightarrow_{in}, [1-6]$

$$\vdash \neg \mathbf{A} \Box A \Rightarrow \mathbf{E} \Diamond \neg A \tag{4.2}$$

1. $x : \neg \mathbf{A} \Box A$	*assumption*
2. $x : \neg \mathbf{E} \Diamond \neg A$	*assumption*
3. $x : \neg \mathbf{E} \Diamond \neg A \Rightarrow \mathbf{A} \Box A$	*theorem* (4.1)
4. $x : \mathbf{A} \Box A$	$2, 3, \Rightarrow el$
5. $x : \neg\neg \mathbf{E} \Diamond \neg A$	$1, 4, \ \neg_{in}, [2-4]$
6. $x : \mathbf{E} \Diamond \neg A$	$5, \ \neg_{el}$
7. $x : \neg \mathbf{A} \Box A \Rightarrow \mathbf{E} \Diamond \neg A$	$6, \ \Rightarrow_{in}, [1-6]$

Hence, it is possible to use the derived rule $\neg \mathbf{A} \Box$: '$from \ \neg \mathbf{A} \Box A \ infer \ \mathbf{E} \Diamond \neg A$.

Now it is shown that the following (induction) rule, which are used in the example verification in the text, is indeed derivable, where $j \notin M1$ and $j \notin Lab^S_{Rigid}$

$$\frac{i : A, \quad (i \preceq j)_\alpha, \quad j : \mathbf{A} \Box (A \Rightarrow \mathbf{A} \bigcirc A)}{i : \mathbf{A} \Box A} \tag{4.3}$$

1.	$x : \mathbf{A}\,\Box(p \Rightarrow \mathbf{A}\bigcirc p) \wedge p$	assumption
2.	$x : \mathbf{A}\,\Box(p \Rightarrow \mathbf{A}\bigcirc p)$	$1, \wedge_{el}$
3.	$x : p$	$1, \wedge_{el}$
4.	$x : \neg\mathbf{A}\,\Box p$	assumption
5.	$x : \mathbf{E}\Diamond\neg p$	4, *derived rule* $\neg\mathbf{A}\,\Box$
6.	$x : \mathbf{E}\Diamond\neg p \Rightarrow \mathbf{E}(\mathbf{true}\,\mathcal{U}\,\neg p)$	*theorem*
7.	$x : \mathbf{E}(\mathbf{true}\,\mathcal{U}\,\neg p)$	$5, 6, \Rightarrow_{el}$
8.	$x : \mathbf{A}\,\Box(\neg p \Rightarrow \neg p)$	*theorem*
9.	$x \preceq v$	assumption
10.	$v : p \Rightarrow \mathbf{A}\bigcirc p$	$2, 9,\ \mathbf{A}\,\Box_{el}$
11.	$v : \neg\mathbf{A}\bigcirc p \Rightarrow \neg p$	10, *rule contraposition*
12.	$v : \mathbf{E}\bigcirc\neg p \Rightarrow \neg\mathbf{A}\bigcirc p$	*derived rule*
13.	$v : \mathbf{E}\bigcirc\neg p \Rightarrow \neg p$	$11, 12,\ \Rightarrow$ *transitivity*
14.	$v : (\mathbf{true} \wedge \mathbf{E}\bigcirc\neg p) \Rightarrow \mathbf{E}\bigcirc\neg p$	*theorem*
15.	$v : (\mathbf{true} \wedge \mathbf{E}\bigcirc\neg p) \Rightarrow \neg p$	$13, 14,\ \Rightarrow$ *transitivity*
16.	$x : \mathbf{A}\,\Box((\mathbf{true} \wedge \mathbf{E}\bigcirc\neg p) \Rightarrow \neg p)$	$\mathbf{A}\,\Box_{in}, 9, 15, [9-15]$
15.	$x : \neg p$	$7, 8, 16, \mathcal{U}_{el}$
16.	$x : \neg\neg\mathbf{A}\,\Box p$	$\neg_{in}, 3, 17, [3-17]$
17.	$x : \mathbf{A}\,\Box p$	$\neg_{el}, 16$
18.	$x : (\mathbf{A}\,\Box(p \Rightarrow \mathbf{A}\bigcirc p) \wedge p) \Rightarrow \mathbf{A}\,\Box p$	$17, \Rightarrow_{in}, [1-17]$

4.4 Complexity and complexity reduction

While the initial research on the resolution based verification of the component model specifications [BBB+06] opened a theoretical prospect of developments in runtime reconfiguration, the complexity of the resolution based verification has raised some concerns with the feasibility of applying this method to a full scale component model. Therefore, there has been a need for complexity reduction. Unlike model checking, where the complexity lies in the specification

part, namely in extracting a model, deductive reasoning 'suffers' in the verification process. One of the ways to overcome the problem is to modify the underlying specification language to obtain a lower complexity similar to the linear time resolution framework [DFK07]. The other main and straightforward approach is to limit the actual amount of the specification properties considered thus avoiding the description of all possible combinations of states and functions of the system. Indeed, it is not needed to analyse the inner working of each component, but only its stateful relations with other components and resources.

In this chapter, the application of formal methods in software development are analysed, and the fundamental behaviour of the GCM approach is defined, in order to identify the most appropriate specification language and validation strategy to utilize. Also it has been given mathematical fundamentals of the specification languages, describing the approach to extend the language ECTL$^+$with deontic modalities to benefit from a logic which allows for model updates. Finally it has been analysed methods of formal verification of the specification, weighing their benefits, and given an example for the chosen approaches.

Chapter 5

Formalizing Behaviour of Grid Components

Behavior Protocols is a term used when formally specifying the behaviour in terms of "ordering of method invocation events" [PV02], where the behaviour of a component is specified by its "frame protocol". In other words, when components are composites of subcomponents, we check the compliance between the protocol and the architecture; instead, when dealing with primitive components, the implementation is checked for compliance; the latter being achieved by the use of a model checker [PPK06]. Similar developments utilize other procedures for the formal specification and verification of the inner components [RBBM04], but unfortunately all of these procedures lack a complementary technique to formalize behavioural interaction of components and environment. In this chapter these issues are addressed and methods are analysed outlining the approach taken in this research.

5.1 Formal specification of components

When considering what parts in the GCM can be used for formal specification, clusters of sections of the GCM specification were created, each of which follows specific criteria and can be easily fed into to a table of specification "patterns". The main details below were examined giving a simplified specification in CTL as reference. As an example, it was considered an Application (the outmost component which must be activated first) which contains four main components Comp1 (a composite component with a sub component SubComp1.1) which is the first to be started after the application is started as it is the first and only component, two components CompA and CompB running in parallel from a broadcast of Comp1 (and SubComp1.1 to start in parallel with CompA and/or CompB), and Comp2 a component from the gathercast (a type of interface connector where multiple bindings are 'gathered' to a single interface) of CompA and CompB. A section from the state tree generated from the specification is shown in Figure 5.1.

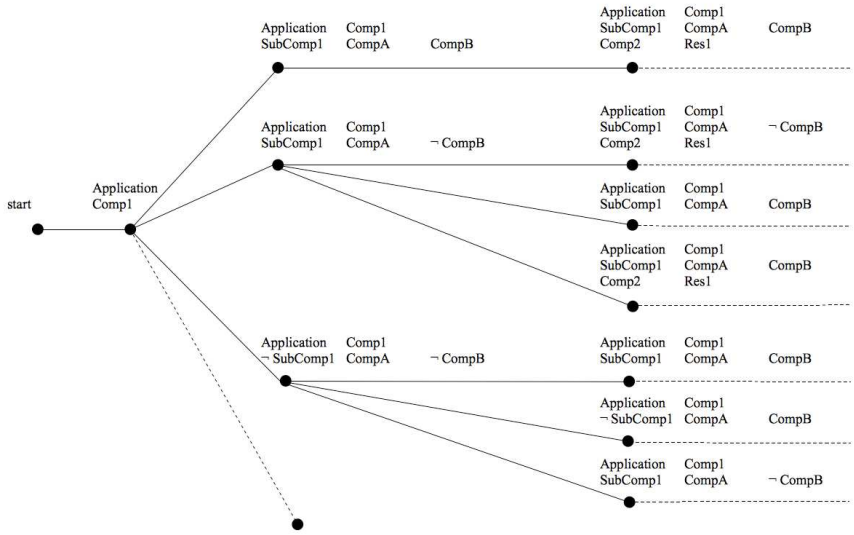

Figure 5.1: States Tree - Sample section

Hierarchical Components Structure. Components in the GCM have a strict hierarchical nature. The application can then be described as: **start** $\Rightarrow$ *Application* and components of the application in the form of: *Application* $\Rightarrow$ $\mathbf{A}\bigcirc Comp1$, *Application* $\Rightarrow$ $\mathbf{A}\diamondsuit Comp2$,

$Comp1 \Rightarrow \mathbf{A}\bigcirc(SubComp1.1 \wedge (CompA \vee CompB))$

Inferring parallel processes from interfaces. When considering interfaces in the GCM, it is possible to group them in two different types: one to one, and broadcast/gathercast. In the former there is a simple connection of one server interface to a client one, while in the latter there is a single server interface which can be bound to multiple client ones and vice versa.

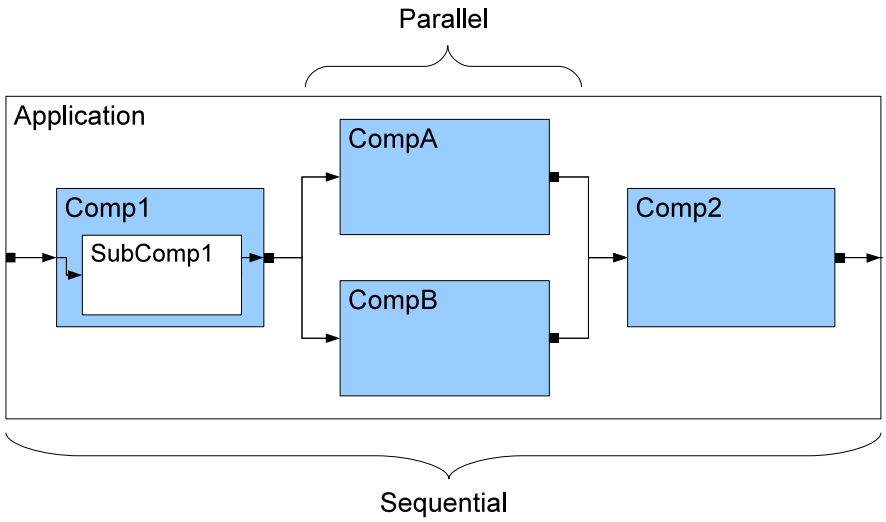

Figure 5.2: Parallelism and Sequential Processes

In either case, interfaces can be very useful to determine whether the communication between components is carried out in a sequential or parallel way. Imagine a component with a broadcast server interface (or several one to one server interfaces): it is possible to easily assume that the components at the client side of those interfaces can be run in a parallel way. On the other hand, a component which has only one server interface, can only run in a sequential way with the component on the client interface side (see Figure: 5.2). Sequential specification looks like: $Comp1 \Rightarrow \mathbf{A}\Diamond Comp2$ while Parallel specification looks like: $Comp1 \Rightarrow \mathbf{A}\bigcirc(CompA \vee CompB)$, etc.

When in a sequential process it is easy to understand that a component will be started sequentially, in a parallel process, there is no real certainty - components might be all started at the next step, or first one and then the others, or perhaps none; all the possible scenarios must be

therefore considered.

5.1.1 State Behaviour of components

The basic lifecycle of components and managed resources, as outlined in 3.2.1, is defined by the states, allowed transitions and operations shown in Figure 5.3.

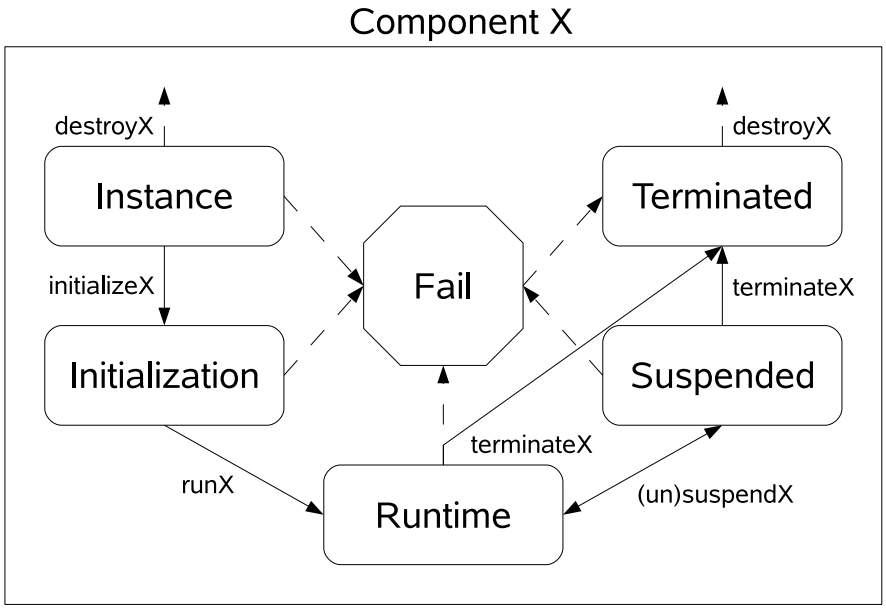

Figure 5.3: Component's Lifecycle States

As a component is such that it conforms to a set of defined states, and to the GCM, it is possible to therefore consider composite components as components that inherit the same properties and conform to state composition. The analysis of the components' instances becomes now crucial. When a component is in the **instance** state, this component (and all its requirements) will be deployed to the appropriate system, and any operations will be performed that are part of the component's instantiation process. This state also presumes that whatever activation is required in order for the resource handler of the component to be valid has been performed. As shown in the diagram, the *initialize* and *destroy* state change commands are supported in this state. The component will then move to **initialization**, where it will wait until a call is made to run; passing on to the **runtime** state, which indicates that the services

provided by the resources that are being deployed are available for use. This state does not indicate any information regarding the operational capabilities of the service, only that it has completed initialization and not failed. At any time, state actions may not complete correctly or the service itself may fail. In response to these failures, the component will transition to the **fail** state. The component may remain active in the system, but its managed resource is presumed to no longer be operational. Once the component is running or has failed it should either eventually or immediately be terminated to stop its services. The **terminated** state represents a state where a component is no longer running and cannot be returned to the running state without redeployment. This state, however, does not eliminate the resource from the system. Upon invocation of the *destroy* command, the component's corresponding resource will be freed. In a system with multiple components, the lifecycle of the whole system is defined by the relationships between the individual component lifecycles. The state of each component is bound to the state of the components it relies on. The hierarchy of the system defines relationships where related components' lifecycles are linked. The component model and the ADL specification help define explicit semantics for guiding lifecycle transitions.

Suspended state

Further analysis should be considered into the **runtime** state above. The suspended state is considered a special state in which the components my be transitioning to and from the runtime state. In this particular state, called the **suspended** state, special attention has to be paid to the states of the resources relative to the component in question (ie. the resources may be released while a component is a suspended state). These factors help refine the way components and relative resources are handled in respect to their stateful behaviour.

Wait state

The case of the **wait** state is a very particular one. This particular case is often referred to when a component is ready to receive the input required for continuing its process (although

some other special cases could arise depending on the specific component). This state often fails back into the generic **runtime** state, since resources are not released by the component although they may not even be "used" (eg. the component may be deployed on a node but not utilizing the processing power). This research is currently forced to consider this state as a particular case of **runtime** state as there are no implemented ways to monitor this situation through the lifecycle controller.

GCM limitations

Although the GCM allows for all theoretical states described in Figure 5.3, introspection in these states is limited to only **Started** and **Stopped**, the first grouping runtime states, while the second describing the terminated or not yet instantiated state. Throughout this research, all states have been considered in order to give a more clear picture of the potential use of the full set; however, during prototype development, it has been possible to utilise only the two exposed states. Nevertheless, it is important to clarify that his 'simplification to two states still leaves plenty of material for investigation in the context of the thesis work, and does not invalidate the proof of concept developed, which will be demonstrated to be functional with these two basic states present.

5.2 State mapping

When considering the state of components and resources in a GCM model, and the runtime monitoring of the environment, we analyse the following introspections.

- For components, by accessing the `LifeCycleController` interface it is possible to know the state of the requested component (namely **Started** and **Stopped**, which relate to the Runtime and Terminated states above).

- For resources, it is possible to monitor their availability status as long as these resources are specified during composition by some deployment descriptor, or at runtime some

metadata provider. As the former is mandatory when using some specific components [ACD+08], it is not mandatory for all. It is assumed that if the developer is interested in using this formal specification for safe reconfiguration of components, he will provide some accessibility to metadata information on runtime availability (as well as list of required resources for the corresponding components), which can be monitored at runtime.

5.2.1 Types of mappings

Each component is mapped at runtime to one or more resources, whether because it is simply deployed to a specific node in a resource, or because such a resource is required by the component to function. Understanding this mapping is helpful when formalizing components and resources by giving us a picture of invariant relationships. The three main types of mappings are defined below.

The one to one mapping. If a component is deployed to a single resource and does not require any other resource to run correctly, this is defined as a one to one mapping. This is the simplest scenario, and it will entail in its formalism the presence of just the single component and the single resource.

The big component mapping. If a component is mapped to two or more resources, this is defined as a big component. Such a component is a composite, where its subcomponents have a one to one mapping with different resources, or a component which has more than one resource associated to it (for example when a component is deployed to one resource and requires to be connected to a database which is located on another resource). In this case the formalism will entail the presence of multiple resources with one component (which could always be a composite one).

The big resource mapping. If a resource is mapped to two or more components, this is defined as a big resource. This is often the case as one resource could run several virtual machines, each one running one or more components (for example nodes in a cluster). Much like the big component, the formalism in this case will entail multiple components and only one

resource.

5.2.2 Formalizing mappings

Independent of the type of mapping, the hierarchical structure of both components (composite and primitive components) and resources (nodes and virtual machines) is crucial in order to simplify the way it is possible to formally describe such relations; in fact it is possible to always translate the mappings above as a collection of one to one mappings (with the same component or resource appearing in more than one of these mappings).

State of components. While the states of components could have a wide spectrum of definition points (such as *initialized, started, suspended, terminated, ...*), and we allow for them to be specified as described above, in the GCM only two states are currently available - i.e. *started* and *stopped*. In a way this simplifies further the formalism by representing the specification as: $Comp1$ for a started component, and: $\neg Comp2$ for a stopped one.

State of resources. When considering resources, it is possible to formally specify the environment thanks to information provided in the GCM deployment file as well as other metadata information (such as availability, performance, reliability levels ...) gathered at development time through a development interface. Furthermore the current state of each resource can be monitored at runtime giving a complete picture of the resources at every given moment in time and any components that might be deployed on or requesting the use of the resource. As an example, external resources are defined as: $Comp1 \Rightarrow \mathbf{A}\Diamond Res1$. Deployment resources are defined as: $Node1 \Rightarrow Res1$ and at runtime it is possible to have definitions like: $Res1 \Rightarrow \mathbf{A}\,\Box(Comp1 \wedge Com2)$.

5.3 Dynamic reconfiguration

In this section the specification language based on the fusion of Computation Tree Logic (CTL) and deontic logic outlined in the previous chapter is utilized to represent the properties of the

behaviour protocol. The requirements of the protocol are understood as norms and specified in terms of deontic modalities "obligations" and "permission". Note that the introduction of this deontic dimension not only increases the expressiveness of the system, but also allows to approach the reconfiguration problem in a novel way.

The reconfiguration aspect is an essential one in this research and we argue that the researched approach brings some important novelty compared with other similar formal approaches to specification and verification. It is needed therefore to introduce this notion and give some definitions and descriptions on how to tackle the problem.

Static and Dynamic Configuration.

- *Static configuration* in a component model is defined as the hierarchical structure of the components and the specific binding of interfaces which connects them. As this is a static view of the system, it cannot include the infrastructure which would complete the system - for example the resources the components will be deployed on. This process is ideal for the application of the static validation of a system, such as model checking.

- *Dynamic configuration* is defined as the process in which the static configuration of the component model is applied to the infrastructure of resources, i.e. the deployment process.

- *Dynamic reconfiguration* is defined as the process in which the mapping of components to resources varies, whether it is the removal of a component or a resource, the addition of one, the change of resources required by a specific component and so forth.

We now focus on the reconfiguration aspect. We refer to reconfiguration as the process through which a system halts operations under its current source specification and begins operations under a different target specification [SK04], both during development and/or deployment (dynamic reconfiguration). Due to the underlying structure of generic Grid systems, the dynamic reconfiguration process is considered as an unforeseen action at development time (known as ad-hoc reconfiguration [BJC05]), therefore programmed reconfiguration is not considered. An insightful example could be the replacement of a software component by the user,

or an automated healing process activated by the system itself. In the case that the system is not yet deployed, the verification of the overall system behaviour (inconsistency check) can be triggered manually at any step of the development process; at this stage it is possible to simply detect inconsistencies and trigger the healing process or complete the verification and return to the user. When the system is deployed, the verification tool will run continuously and the system will report back the current states for model mapping; if a reconfiguration procedure is requested or inconsistency detected, the healing process is triggered.

The dynamic reconfiguration process works in a circular way [Figure: 5.4] and it is divided into three majors steps: model update request (where a request from a user or the system itself is fired), model mapping (where a snapshot of the current state of the system is captured) and finally the healing process (where the actual reconfiguration takes place). The approach here is to specify general invariants for the infrastructure and to accept any change to the system, as long as these invariants hold. We assume that the environment has some pre-defined *set of norms* which define the constraints for the system, in order to ensure system safety, mission success, or other crucial system properties which are critical especially in distributed systems.

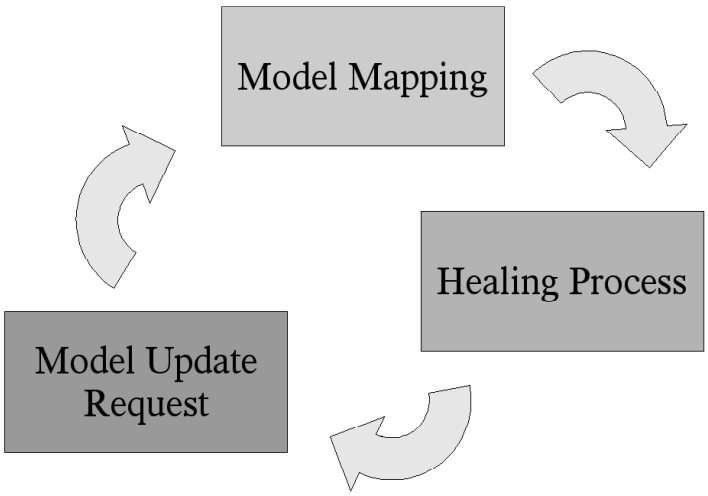

Figure 5.4: Reconfiguration Cycle

5.3.1 Model update

A model update request can be triggered by a user's intention to reconfigure the system, or by an inconsistency detection from the verification tool. It refers in the model as a change to the behaviour specification and it is constrained by the infrastructure restrictions. For example, the user might want to upgrade a component, but these changes must conform to the limitations set for such component. If the changes themselves are safe for the system, the tool passes to the next step. For the verification process to understand its current state in the temporal tree, there is a need for a constant 'model mapping'; in other words, a background process needs to be present in order to map the structure of the system into a model tree. This can be easily implemented alongside a current monitoring system which will keep track of this mapping indicating which parts of the system are currently in which states in the model tree [BCH+02]. This process is essential to ensure that no 'past' states are misused by the tool during the healing process.

If the model behaviour needs to be updated according to the new external input, parts of the system specification need to be changed. This process is the key to this type of model update architecture and is necessary because, unlike model revision in which the description is simply corrected but the overall system remains unchanged, by updating the specification we are fundamentally changing the system by adding, deleting and replacing states in the model behaviour [EG92]. Here different types of changes are dealt with in a similar fashion, independently from the origin of the update (external user input or self healing process). The behaviour specification is 'extended' to a new type of specification and the verification process is resumed from this point forward [Figure: 5.5]. This model update process consists of:

(i) Norms/Invariant check. Utilise norms and invariants in the specification for constraints on the set of states to be updated. Here it is possible to detect the deontic properties in the specification which could be utilised in the healing process.

(ii) Compatibility check. Check if the supplied update to the model conforms with the

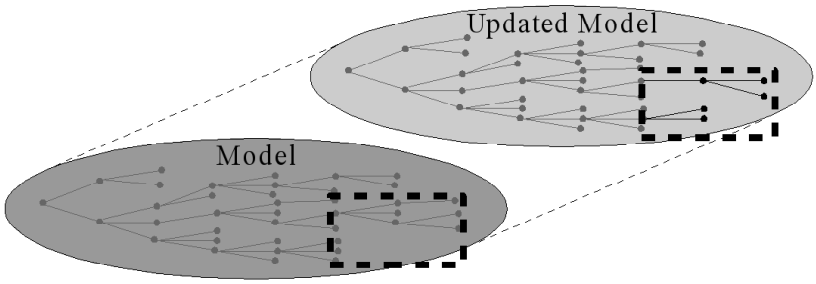

Figure 5.5: Model Update

the set of states to be updated, in other words, the system must check for the presence of the standard bindings of the components, controllers, etc; if so, the model is updated, otherwise, the healing is triggered.

(iii) Healing process. Search the tree model for a set of states which conform with the norms and invariants, and is applicable for this set of states. Note that candidate states for such an update in relation for some state s_i, do not have to be in an 'achievable' future of s_i, i.e. do not have to belong to a subtree with the root s_i, but only have to be 'accessible' from the current state according to the norms set by the infrastructure. The candidate set of states (or a more readable parsed version) is reported to the user/developer as a possible solution to the inconsistency detected. (Note that healing is also triggered if there was no supplied update as in the case of inconsistency detection).

In this chapter it has been analysed the researched approach in the specification of behavioural interaction of component models and the environment they sit upon. It has been identified how to specify the state behaviour of components without having to consider the compliance of their implementation (which can be left to other methods), and has been defined how the running states of deployed components and the resources they utilize can affect the dynamic reconfiguration of the Grid system.

Chapter 6

Implementation

In this chapter it is described the process involved in building the prototype which allows to demonstrate the capabilities of the methods and techniques described in this thesis, completing the author's contributions in this research. An overview is given of how the prototype is utilized and integrated in the related work of the Grid development environment, the features and capabilities of the tool developed, and a how the prototype was tested against use case scenarios.

6.1 Strategies

When implementing a prototype to demonstrate the capabilities of the method researched, the focus has been on streamlining a process of formal specification and verification which would be integrated in the development environment for the programmer, as well as transparent to the grid user, and most importantly, as automated as possible. The ideal approach to such a development would have been in the case where the underlying structure of the development process would be based on logic, such as in a project like [Bol05]. Unfortunately, such a system has never been developed before, and would have been an impossible task to accomplish for this research due to the fact that an entire framework needs to be clearly defined and constructed.

The focus has been shifted to the tools present at the time of development, and making the most of what was available to construct a working prototype. The implementation process began by analysing what was already available, and developed the tool around it. The first step was to construct a method for the formal specification of the behaviour of states of the system. Secondly, an appropriate tool for the verification of the specification needed to be plugged-in, in conjunction with an engine to allow for model update and handle reconfiguration requests. The task was divided into three parts: the 'static' specification tool in the development environment; the Monitoring engine, which would handle reconfiguration requests, state mapping and verification responses; and the Verification engine. In figure 6.1, it is possible to see how this prototype has been constructed, the blue parts denoting prototype developments, and the grey parts denoting third party developments.

The intended functionalities of the verification tool implemented are as follow:

- To provide a formal specification of the dynamic application, based on it state transitions. This includes hierarchical components, interfaces, bindings and controllers.
- To construct a formal specification of the grid environment, including (but not limited to) resources upon which the components run, external resources and services, and any other real or virtual parts which are needed by the grid application to perform correctly.
- To create a verification engine capable of prove the validity of the application in respect to the grid environment it relies upon.
- To automate the process of reconfiguration in a safe and stable manner.

Furthermore, the requirements of the tool for it to integrate correctly with the development and deployment environment are as follow:

- A fully constructed grid application, created using the Grid development environment.

- A deployment file describing the resources available and the initial location of the grid application's component on these resources.
- External resources specified through the Grid development environment, defining resources not present in the deployment file which might be needed by the application to run correctly.
- A monitoring and steering application which exposes the states of components and resources at runtime and allows for their reconfiguration.

It is important to note that although this prototype has been implemented to work side by side with the Grid development environment, it can be modified to work with any other similar application, as long as it exposes the same set of basic functionalities.

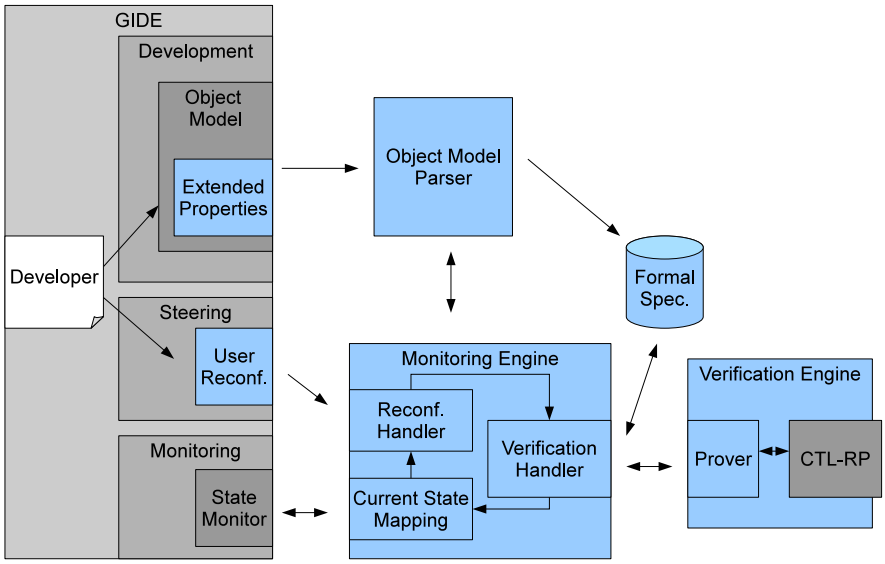

Figure 6.1: Prototype Structure

6.2 The GridComp IDE

Since it was possible to collaborate with the GridComp project on the development of the prototype for this research, it was formally included in the specifications of the GridComp IDE project as a non-functional plug-in (a structure of the project can be seen in Figure 6.2 where

'Verification Tool' denotes the plug-in). The development environment has been implemented as a way for grid developers to take advantage of a graphical interface to easily construct Grid application by making use of a selection of tools. The GridComp IDE (GIDE) is essentially a plug-in in itself for the Eclipse environment [Fou09], which allows to graphically construct a component model of the grid application, as well as adding functionality to the components through the use of the ProActive API [BBC+06]. Furthermore, the GIDE provides the user with a deployment view that allows to specify and run the application on a Grid environment, as well as a monitoring and steering tool for monitoring of the application as well as dynamic reconfiguration of its structure.

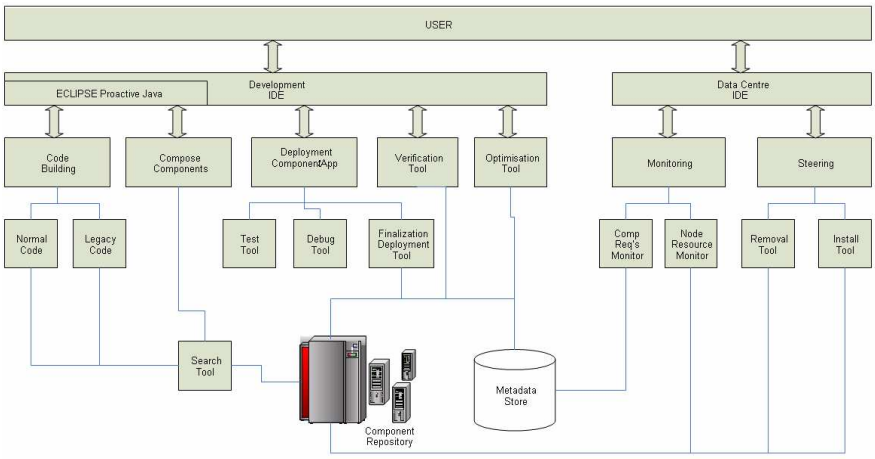

Figure 6.2: GIDE Structure

6.3 Verification tool Features

The Verification tool was developed as a plug-in for the GridComp IDE, and as such it was able to utilize anything that could be provided through its interface or API, but it should not modify any part of its pre-existing structure. As such, the development started by selecting which built-in tools in the GIDE development could be utilized for the development of this prototype and which part would have to be developed or whether a third party tool would be required. In this section it is outlined its main features and code snippets are added to describe

essential procedures of the prototype.

6.3.1 Object Model Parser

The Object Model Parser is a tool developed in this research which allows to make use of the GIDE Object Model, which is a view of the structure of the Grid application, which includes the hierarchical structure of components given from the ADL, their interfaces and bindings. The parser translates the Object Model into a language understandable by the verification engine, i.e. the formal specification developed in this thesis. Furthermore, the parser can create a specification also from the 'extended properties view' - a part of the GIDE development environment which allows the developer to define extra connectivity characteristics of components and resources, effectively further extending the details in the specification. The parser makes use of hard coded patterns to match generic specification formulae to specific structural information on the object model of the application.

Components Hierarchy

As each component is bound by a strict hierarchical nature which defines the order by which each component should be instantiated and terminated, started and stopped etc., it is important to ensure that the appropriate specification is included which would restrict a reconfiguration from, for example, stopping a component before its subcomponents are stopped as well.

Listing 6.1: Method processing the components' hierarchy (partial code snippet)

```
/* This method creates formulae relative to components hierarchy */
public void processObjModComp(ComponentDesc root, Specification spec
    ) {
 if (root == null){
  //System.out.println("WARNING: root ComponentDesc empty.");
  return;
 }
```

```
root = ComponentDesc.unwrap(root);
List<ComponentDesc> outerCompsList = new ArrayList<ComponentDesc>()
    ;
outerCompsList.add(root);
List<ComponentDesc> innerCompsList = new ArrayList<ComponentDesc>()
    ;
spec.addToCompSpec(this.addStartImp(root.getName()));


while (outerCompsList.size()>0){
 for (int i=0; i<outerCompsList.size(); i++){
  List<ComponentDesc> tempInnerCompsList = new ArrayList<
      ComponentDesc>();
  tempInnerCompsList = outerCompsList.get(i).getInternalComponents
      ();
  if (tempInnerCompsList.size()>0){
   String tempSpec;
   tempSpec = this.appendImp(outerCompsList.get(i).getName());
   for (int j=0; j<tempInnerCompsList.size(); j++){
    String interName = tempInnerCompsList.get(j).getName();
    tempSpec = tempSpec + interName;
    innerCompsList.add(tempInnerCompsList.get(j));
    if (j<tempInnerCompsList.size()-1){
     tempSpec = this.appendAnd(tempSpec);
    }
   }
   spec.addToCompSpec(tempSpec);
```

```
    }
    else{
     innerCompsList.clear();
    }
    tempInnerCompsList.clear();
   }
   //clear outer component and replace with first inner
   if (innerCompsList.size()>0){
    outerCompsList.clear();
    for (int i=0; i<innerCompsList.size(); i++){
     outerCompsList.add(innerCompsList.get(i));
    }
   }
   //we have reached primitive comp
   else{
    outerCompsList.clear();
   }
   innerCompsList.clear();
  }
}
```

Interfaces and Bindings

Similarly to the hierarchical nature of the components, their connections through bindings between interfaces needs to be addressed during the formal specification. This process can give an insight on the communication flow as well as on parallel execution of components.

Listing 6.2: Method processing the components' bindings and interfaces (partial code snippet)

```
/* This method creates formulae relative to interfaces and bindings
   */
public void processObjModIntfBind(ComponentDesc root, Specification
   spec){
 List<ComponentDesc> compList = new ArrayList<ComponentDesc>();
 if (root == null){
  //System.out.println("WARNING: root ComponentDesc empty.");
  return;
 }
 root = ComponentDesc.unwrap(root);
 compList.add(root);
 List<InterfaceDesc> intfList = new ArrayList<InterfaceDesc>();
 List<BindingDesc> bindList = new ArrayList<BindingDesc>();


 //process comps list
 for(int i=0; i<compList.size(); i++){
  intfList = compList.get(i).getInterfaces();
  //process interfaces of comps
  for (int j=0; j<intfList.size(); j++){
   String intfName = intfList.get(j).getName();
   //found interface
   bindList = compList.get(i).getInterfaceBindings(compList.get(i).
      getName(), intfName);
   for (int k=0; k<bindList.size(); k++){
    String bindName = bindList.get(k).getLabel();
    //found binding
```

```
    }
    bindList.clear();
  }
  intfList.clear();
 }
}
```

6.3.2 GIDE Extended properties View

An important step in the development was to identify parts of the Grid architecture which were lacking in detail of resource connections to components. All the information regarding the resource structure in fact is included in the deployment file which is loaded before running the Grid application. This file contains information about the resources where the components are to be deployed, such as nodes and virtual machines, but it lacks information on other resources which might be required, for example a database or a service. A view was created in the GIDE which allows the developer to specify such occurrences (see Figure 6.3), which is then automatically parsed to create a suitable specification at deployment phase.

Listing 6.3: Method processing the external resources (partial code snippet)

```
/* This method creates formulae relative to external resources,
   services, etc */
public void processExternal(DB myDB, Specification spec){
 List<String> compList = myDB.getComponentNames();
 //process compList from externals database
 for(int i=0; i<compList.size(); i++){
  for(int j=0; j<myDB.getExtensions(compList.get(i)).size(); j++){
```

```
    String mySpec = this.appendObligation(this.appendImp(compList.get
        (i)))
    +this.appendAnd(myDB.getExtensions(compList.get(i)).get(j).
        getType())
    +myDB.getExtensions(compList.get(i)).get(j).getExt();
    spec.addToExtSpec(mySpec);
    //System.out.println(mySpec);
   }
  }
}
```

6.3.3 Formal Specification Database

The Formal Specification Database was developed as a concrete file, instead of a volatile instance, in order to be as versatile as possible. It is in fact possible that the Grid application developer and the user may be two different people, making it difficult for the application developed (and therefore the specification created during the development) to be run right after its implementation. The specification had to be made as a concrete part of the application which could be moved across different machines similarly to the way other files created by the GIDE could. It was decided that the best location for storing this file would be alongside other files defining the structure of the developed grid application, while ensuring that its presence (or absence) would not conflict with the normal functioning of the GIDE.

6.3.4 GIDE Monitoring and Steering

Monitoring and steering is an important component in every Grid infrastructure [AAB+04]; unfortunately, this feature was not fully completed in the GIDE, and some important aspects, such as support for skeletons [ACD+08], was an integral part of testing for the prototype

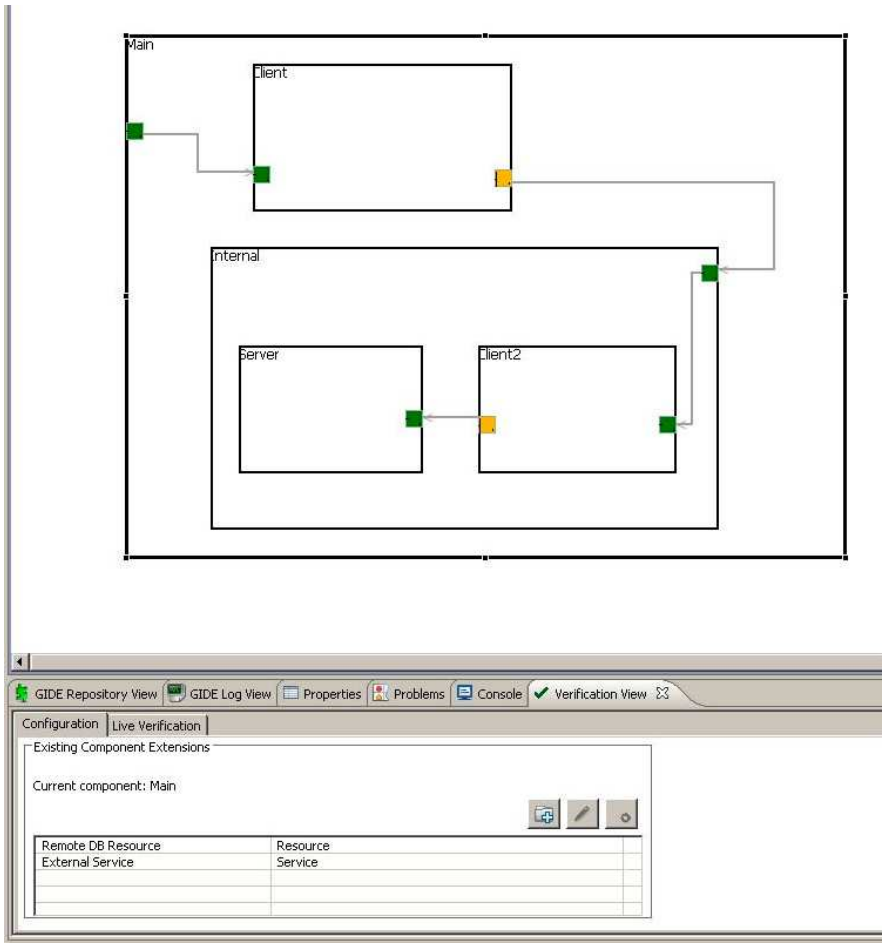

Figure 6.3: GIDE Verification View

developed. For this reason, all the possible features integrated within the GIDE were developed, such as calls to the monitoring view, but were left out from the final prototype and instead it was opted for a simulation environment, where better control over steering was enabled. The implications for the results of using a simulated environment over a fully integrated system are minimal in this case, as the prototype has been developed to allow for the same features, while, from testing on the system, it has been tested that monitoring and steering does not impact in a considerable way on the verification and reconfiguration process, the most important parts being the validity of the specification reported and the time taken to run the verification itself.

6.3.5 Monitoring Engine

The Monitoring engine uses calls to the State monitor in the GIDE through the *getFcState* interface of the *LifeCycleController*. This in return gives the state of components, in terms of *started* and *stopped*. Through this, two integral parts of the current state of components can be inferred:

1. If the component is in a *started* state, the component is in use, and therefore its subcomponent and the composite component it sits within, are directly affected by its status. It is possible to stop this component only if its subcomponents (if any) are stopped first, and this is a recursive behaviour. This in turn affects its composite component (which always exists apart from the main composite component, i.e. the application), which cannot be stopped unless the component in question is stopped first.

2. If the component is in a *stopped* state, it might never have been initiated, it may be terminated, or might have stopped its functionality to be restarted at a later stage. In this case, the prototype would add specification that the component in question must not start until a possible reconfiguration of such component would be finished, so that the Grid application would simply wait for the component to start after the reconfiguration takes place. A possible extended implementation on the suspended / wait state might be developed assuming that if the component has never been started, it is not in the suspended / wait state. This kind of insight would be useful in the case that the component utilises some external resource and would activate them at initialization - if that is the case, the resource might be active while the component appears to be stopped.

A first run of the verification takes place after deployment, to ensure correct initial configuration. After that, the Monitoring engine awaits calls from the GIDE Steering (simulated in this prototype development) for reconfiguration; these calls might be initiated externally by the user, or internally by the tool itself. If such a call is fired, a new mapping of the current states is called through the GIDE monitoring and the new set of specification is passed to the

Verification Engine. In both cases (initial configuration and reconfiguration), the Monitoring engine will handle the result of the verification and pass an appropriate message to the GIDE event handler, and, if allowed, perform the reconfiguration. The reconfiguration is left to the Steering framework of the GIDE and is outside the scope of this research, and will not be analysed. However, if the reconfiguration cannot take place because of a failed verification, the message passed to the GIDE event handler will reference to the portion of the verification that failed, giving an insight to the user on the reasons why the reconfiguration cannot take place.

6.3.6 Verification Engine

In the Verification engine, two main components can be identified, a handler and a theorem prover. The handler essentially converts any part of the specification which cannot be understood by the prover - if any (as it is a third party tool) and acts as a sort of debugger in case that the prover malfunctions. Furthermore, it interprets the results from the prover and relays them back to the Monitoring engine. The prover utilized is CTL-RP [ZHD08] developed by L. Zhang at the Unversity of Liverpool. This is as of time of writing, the only fully implemented resolution based prover for CTL available; it accepts as input formula in the form:

$$AGp <-> EF\ pand(implies(AG(p), not(EF(not(p)))), implies(not(EF(not(p))), AG(p)))$$

As this prover does not support language $\mathrm{SNF}^{D}_{\mathrm{CTL}}$ with deontic modalities, this is left to future developments. There is no impact on the specification from this - only the automated reconfiguration process is limited by it. It is possible in fact to prove the validity of a possible reconfiguration scenario using $\mathrm{SNF}_{\mathrm{CTL}}$ and returning to the user a notification if the reconfiguration will not be able to proceed. Recall that deontic modalities are in fact used in the case of a failed reconfiguration to search for a similar scenario in either suggest possible changes to the user based on the similar scenario, or if possible, apply the same set of rules, and proceed with the reconfiguration automatically (healing process). With this it is possible for the system

to identify common traits in the way that the application performs during a reconfiguration, and apply the same set of rules in a similar process. Consider for example that the user has added a new resource to the system which requires a specific component to be always running; if these requirements are specified beforehand, the reconfiguration can take place without the need for adjustments, but if it is not, the user would normally have to stop the entire application in order to proceed set the requirements in place. If on the other hand the new resource is given in a specification that allows for the required deontic modalities, the tool would now be capable to automatically find a similar structure in the system, and perform the reconfiguration autonomously.

As there is currently no implemented prover for $\mathrm{SNF}^{D}_{\mathrm{CTL}}$, the healing process cannot be demonstrated in the prototype, however as the CTL-RP prover offers the possibility of seeing the complete proof of the verification process, this allows the handler to interpret any failure and report relevant parts back to the Monitoring engine, and the reconfiguration can be done manually. A complete proof of the formula above can be found in Appendix A.

6.4 Use Cases and experimental work

In testing the tool, it was considered the use cases described in [faEIG]. This use case was developed by IBM for testing the developments in the GridComp project and the GIDE [GBT+07]. It seemed only fitting to utilize one of these use cases for the testing. The IBM biometric identification use case's (BIS) objective is to demonstrate a sample biometric identification system which requires the computational power of a Grid application due to the fact that it would be applied to a large set of fingerprint biometrics connected to a wide user population. It is constructed using GCM components and makes use of particular components constructs - skeletons [ACD+08] - which was important to include in testing. A complete report on the developments related to this use case can be found on [WADG09]. An overview of the BIS

architectural design is given in Figure 6.4 (figure taken from [WADG09]).

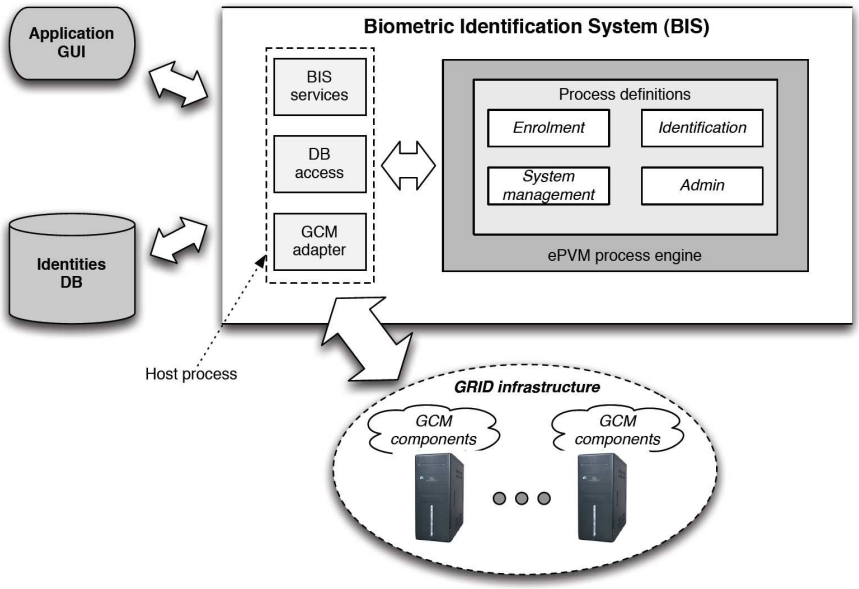

Figure 6.4: BIS Architectural Design

6.4.1 Testing

Although testing on the use case has been performed internally by the GridComp project, these particular results were not of great interest, as they do not apply to a (re)configuration scenario. However, the simulation of a run of the application had been applied to the prototype for constraints on the dynamic reconfiguration timing (details on the reconfiguration of this use case can be found on section 5 of [WADG09]). Results given below are therefore based on the specification and verification, plus the reconfiguration, of the use case, and as this is a simulation, should not be taken as real world settings, but as indicative figures. In testing the prototype, the specification was constructed automatically from the given component model in Figure 6.5. The component model comprises of multiple parts, but the ones of interest to the research are the BIS handler, in charge of sending the identification requests, and the workers (referred in the model as IDMatchers), in charge of running the comparison with the information retrieved from the identity database. The workers are enclosed by a composite

component, and their number can change depending on available resources. In the component model figure below the identity database can be seen; but, because of the its limitations, this is not the case in the ADL, therefore it was added as an external resource connected to each IDMatcher component present. This allowed to test the scenario in which a required resource was not present during (re)configuration. The formal specification of the component was tested in multiple variations, some were created from the full object model with skeleton construction (the farmer/worker construction in which the creation of IDMatcher components is automated by the skeleton), some from full object model without skeletons (manual creation of IDMatcher components), some not considering the database as an external , etc. The range of the tests spans across multiple parts, but was focused on the following points:

- Quality of the formal specification
- Speed of formal specification process (runtime only)
- Speed of formal verification process
- Quality of verification results
- Processing power requirements of the tool

Please note that the speed of the creation of the formal specification has been measured only during runtime, i.e. the part of the specification relevant to the current state of the resources and eventual new components introduced. The speed by which the formal specification of the object model and environment was constructed is not of particular interest as it happens during development, and does not impact on reconfiguration or other time sensitive processes.

6.4.2 Results

Before detailing the results from the testing, the timing on which the simulator was running should be outlined, following the results detailed in [WADG09], and assuming average, or near to, scenario. Although most of the results provided below are from a simulation environment,

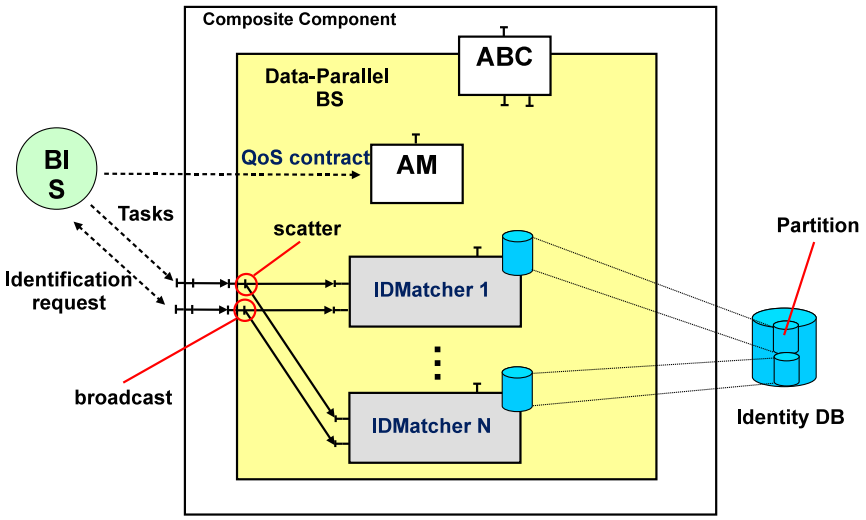

Figure 6.5: BIS Component Model

the utmost care to follow real settings has been preserved, in order to give the best indicative results possible. In order to tune the system, the considered parts were: speed by which each worker is able to process biometric information, and timing of workers reconfiguration in a grid setting. For the first part, it was reported that on a normal laptop CPU, a worker was typically able to process about 1000 fingerprints per second (assuming that all data was previously stored in RAM). As for the time required for reconfiguration, the BIS tool was tested on Grid5000 [Pro08] and was reported that every reconfiguration operation took about 9 seconds including a complete redistribution of the database between the workers. The tool was tested with 50 workers and 50000 fingerprint database, meaning that for the given database size, each identification request required about 10 seconds to be processed. With this knowledge in hand, it was possible to gather the following results in terms of timing based on a normal laptop CPU. Note that the third scenario considered has a slightly greater complexity in terms of number of components due to the structure of the skeleton configuration, which adds extra functionalities to the application and therefore making the specification more complex. Although this approach adds more computation on the verification, it should be noted that it also adds more introspection in the application's workflow, allowing for a potentially safer and more accurate reconfiguration of components.

- For the simplest specification, considering a non-skeleton based scenario, 53 composite and primitive components (including bindings and interfaces) and no external resource, the runtime specification process was completed on average in less than 0.1 seconds and required 0.2% CPU power to be generated. In terms of verification of the given specification, the time required was about 2 seconds and around 1% CPU power.

- For the slightly higher level of specification, which changed from the previous by adding the database as external resource, the runtime specification process was completed on average in less than 0.1 seconds and required 0.2% CPU power to be generated, just as above. For the formal verification, the time required was just over 2 seconds and 1% CPU power.

- Finally, for the specification which considered the skeleton based scenario, 57 composite and primitive components (including bindings and interfaces) plus the external resource, the runtime specification process was completed on average in less than 0.2 seconds and still required 0.2% CPU power to be generated. In terms of verification of the given specification, the time required was about 5 seconds and around 2% CPU power.

Note that processing power is given here as reference but might not play an important part as generally the verification tool would run on the main machine and not be distributed to a node, or it might not be an issue altogether if the verification tool is run on a separate machine. From these results, in terms of the reconfiguration process, an extra 2 to 5 could be added on top of the 9 seconds required per reconfiguration as tested in [WADG09]. The use of the tool might add an extra 40 seconds to the initial deployment if each addition of a worker (7 workers were automatically added by the skeleton structure) is tested for consistency. On the other hand, if a malfunction during the redistribution of the database is simulated with the tool, the prototype is able to detect it and halt the reconfiguration, pointing to the database as the culprit. The quality of the specification (and therefore the verification results as those are linked to the initial specification) have been satisfactory in general, although they lack the

deontic modalities which would allow for autonomous healing. In more than 20 tested scenarios (10 successful reconfiguration and 10 unsuccessful), the tool was able to always produce the correct specification, and point to the component or resource which was the culprit for the unsuccessful reconfigurations.

In this chapter it has been demonstrated how the research in this thesis can be implemented in a prototype, and proven that it can be successfully used in a (re)configuration scenario. The structure of the implementation has been outlined, as well as details of its working parts and their correlation to the theoretical fundamentals. The implementation was also tested on a grid application use case, and the results obtained are reported, outlining possible improvements that could be applied in the future to newer versions of the prototype.

Chapter 7

Conclusions

In this thesis, many points have been touched, from formal specification of Grid component models, to deductive verification of distributed applications, to safe reconfiguration of Grid systems. All these points together though, allow for a different prospective in the way formalism is conceived. When talking about behaviour formalism of applications, it is common to think as to whether the software is performing the given task correctly, disregarding the side effect that the application might have on the environment. These prospectives in fact should be complementary: allowing for both aspects to be thoroughly analysed. As the developments into the formalism of software applications have been already successfully implemented with various techniques, the focus of this research has been on the less researched part of environment formalism. It has been discovered that the infrastructure of a software application, especially a distributed one, plays an important role in the lifetime of the application itself, restricting and driving its behaviour in ways that can easily influence aspects such as dynamic reconfiguration. It has been determined that, while the component models which could be used to construct the application offer a series of advantages during the development process, it lacks in safety functionalities during dynamic reconfiguration of distributed applications, making this the perfect scenario to apply an approach to environment formal specification and deductive verification.

In order to do this though, the critical aspect of integration had to be analysed: it is important in fact to determine the way by which a process of software formalism can be combined with the one of software development, which traditionally have not been developed in parallel.

Contributions Details

In this thesis it has been analysed the process of integration of models abstraction for formal specification with distributed applications and their environment. This approach is a novel way to address integration which has never been conceived before in the realm of Grid systems. It allows for a deeper introspection into the structure and interoperability of components and resources, making this technique essential when considering formal tools in developments. We have in fact demonstrated that this procedure is ideal for complementing techniques, such as model checking, because it considers an abstraction of the system which is not part of the one used by these tools (such as the working code abstraction), and, because it does not interfere with that domain, it can be simply run in conjunction.

This thesis contributes in the aspect of component-based Grids development by identifying aspects of the development of Grid applications which are not normally considered influential in the safe running of distributed applications. The environment on which a Grid application lays on is assumed to be able to handle unforeseen circumstances of the applications it hosts; and while the environment might be able to recuperate from these occurrences, it is not always the case that the application will. During this research it has been outlined which parts of the Grid system have to be considered in order to prevent this, and illustrated a technique that can be used to store and use this knowledge about the system to our advantage when dealing with failure prone scenarios.

As a technique for formal specification and verification, this thesis contributes in the area of temporal specification and resolution. In this research it has been developed an autonomous

methodology for, not only clearly specifying the structure and environment of distributed applications, but also for automated healing of the application through the extension of a formal language - ECTL$^+$ - with deontic constraints in ECTL^+_D. This innovative approach has been proven to add a level of introspection into the way formal models can be updated and applied to real world reconfiguration of large scale systems. It is now possible to define constrains in the system which can detect a future failure and trigger an automated healing process, effectively bypassing the user altogether and becoming an invisible process. This can be a great advantage to users of the system who are not familiar with its inner workings.

The dynamic reconfiguration aspects in Grid computing has also been addressed in this thesis, identifying this process as the main contributing factor to resources related failures. It is understood that resources play a significant role in the reconfiguration process of parallel applications, and the availability and reliability of these resources is cause of great concern during reconfiguration as it is often difficult to manually keep track of changes in the system. The issue of dynamic reconfiguration has been a focus point in this thesis, and the creation of a novel technique to enable safe and automated reconfiguration has been a main contribution, since this approach takes into consideration parts of the Grid system which have not been considered before. As the developed approach can consider dynamic reconfiguration, it is also able to consider static configuration, as both are based on the same formalism. Furthermore, the approach described in this thesis is easily transferable to similar distributed systems, as their basic structure is very similar.

In this research a proof of concept has also been implemented in terms of a prototype implementation. This prototype is used to demonstrate the possibility of the approach and its real world applications. It is shown that the implementation can easily integrate with a given development environment, and can be used for the dynamic reconfiguration of components, while maintaining its properties of ensuring the safety of the process.

Future developments

Finally, future developments in the area could include further insight in the use of deontic modalities in order to obtain a better updated model with more insights into future events, eventually leading to a system where the user is not required to do manual changes for certain types of reconfiguration failures. This could be integrated in the prototype, given that a version of the verification tool which supports deontic modalities has been implemented. As the approach provided in this research is easily translated for other types of distributed computing applications, the research should be applied to other developments, potentially discovering alternative views or improved techniques, expanding the current capabilities to include all applicable scenarios.

Appendix A

CTL-RP [ZHD08] Sample Proof

Listing A.1: A sample proof generated by CTL-RP

```
========================================  CTL-RP
    ========================================


=====TWB input:
((AG(p) -> ~(EF(~(p)))) & (~(EF(~(p))) -> AG(p)))
=====ctl:
and((AG(p) => ~(EF(~(p)))), (~(EF(~(p))) => AG(p)))
=====ctl:add negation
~(and((AG(p) => ~(EF(~(p)))), (~(EF(~(p))) => AG(p))))
=====ctl:NNF
or(and(AG(p), EF(~(p))), and(AG(p), EF(~(p))))
=====ctl:SNF


===== clause [1]: implies(start, new0001)
===== clause [2]: implies(T, or(not(new0001), new0002, new0003))
===== clause [3]: implies(new0003, EF 1(not(p)))
===== clause [4]: implies(new0004, AX(new0004))
```

```
==== clause [5]: implies(T, or(not(new0004), p))

==== clause [6]: implies(T, or(not(new0003), new0004))

==== clause [7]: implies(new0002, EF 2(not(p)))

==== clause [8]: implies(new0005, AX(new0005))

==== clause [9]: implies(T, or(not(new0005), p))

==== clause [10]: implies(T, or(not(new0002), new0005))


========== Number of saturation: 1 ==========

Input Problem:

1[0:Inp] ||  -> new0001(0)*.

2[0:Inp] || new0001(U)* -> new0002(U) new0003(U).

3[0:Inp] || new0004(U) -> new0004(app(V,U))*.

4[0:Inp] || new0004(U) -> p(U)*.

5[0:Inp] || new0003(U)* -> new0004(U).

6[0:Inp] || new0005(U) -> new0005(app(V,U))*.

7[0:Inp] || new0005(U) -> p(U)*.

8[0:Inp] || new0002(U)* -> new0005(U).

 This is a monadic Non-Horn problem without equality.

 This is a problem that has, if any, a finite domain model.

 There are no function symbols.

 This is a problem that contains sort information.

 The following monadic predicates have finite extensions: new0001.

 Axiom clauses: 8 Conjecture clauses: 0

 Inferences: IORe=1

 Reductions: RFMRR=2 RObv=1 RTaut=1 RFSub=1 RBSub=1 RCon=1

 Extras     : No Input Saturation, No Selection, No Splitting, Full
     Reduction,
         Ratio: 5, FuncWeight: 1, VarWeight: 1

 Precedence: div > id > app > p > new0001 > new0002 > new0003 >
```

```
        ind_1 > new0004 > ind_2 > new0005 > wind_1p > wind_2p > 0 > ls >
            start > T
 Ordering   : KBO
Processed Problem:


Worked Off Clauses:


Usable Clauses:
1[0:Inp] ||  -> new0001(0)*.
7[0:Inp] || new0005(U) -> p(U)*.
4[0:Inp] || new0004(U) -> p(U)*.
8[0:Inp] || new0002(U)* -> new0005(U).
5[0:Inp] || new0003(U)* -> new0004(U).
2[0:Inp] || new0001(U)* -> new0003(U) new0002(U).
6[0:Inp] || new0005(U) -> new0005(app(V,U))*.
3[0:Inp] || new0004(U) -> new0004(app(V,U))*.
        Given clause: 1[0:Inp] ||  -> new0001(0)*.
        Given clause: 7[0:Inp] || new0005(U) -> p(U)*.
        Given clause: 4[0:Inp] || new0004(U) -> p(U)*.
        Given clause: 8[0:Inp] || new0002(U)* -> new0005(U).
        Given clause: 5[0:Inp] || new0003(U)* -> new0004(U).
        Given clause: 2[0:Inp] || new0001(U)* -> new0003(U) new0002(U).
        Given clause: 9[0:Res:1.0,2.0] ||  -> new0003(0) new0002(0)*.
        Given clause: 10[0:Res:9.1,8.0] ||  -> new0003(0)* new0005(0).
        Given clause: 11[0:Res:10.0,5.0] ||  -> new0005(0) new0004(0)*.
        Given clause: 6[0:Inp] || new0005(U) -> new0005(app(V,U))*.
        Given clause: 3[0:Inp] || new0004(U) -> new0004(app(V,U))*.
================== derived by MRR: 0 ==================
================== derived clauses: 3 ==================
```

```
========== subsumed clauses: 0 ==========

========== Number of saturation: 2 ==========

[...]

========== Number of saturation: 6 ==========

Input Problem:

29[0:Inp] || new0003(U) p(U)* -> wind_1p(U).

30[0:Inp] || wind_1p(U) p(app(ind_1,U))* -> wind_1p(app(ind_1,U)).

31[0:Inp] || new0003(U) p(U)* new0004(U) -> .

32[0:Inp] || new0003(U) p(U)* new0005(U) -> .

33[0:Inp] || wind_1p(U) p(app(ind_1,U))* new0004(app(ind_1,U)) -> .

34[0:Inp] || wind_1p(U) p(app(ind_1,U))* new0005(app(ind_1,U)) -> .

52[0:Inp] || new0002(U) p(U)* -> wind_2p(U).

53[0:Inp] || wind_2p(U) p(app(ind_2,U))* -> wind_2p(app(ind_2,U)).

54[0:Inp] || new0002(U) p(U)* new0004(U) -> .

55[0:Inp] || new0002(U) p(U)* new0005(U) -> .

56[0:Inp] || wind_2p(U) p(app(ind_2,U))* new0004(app(ind_2,U)) -> .

57[0:Inp] || wind_2p(U) p(app(ind_2,U))* new0005(app(ind_2,U)) -> .

 This is a monadic Horn problem without equality.

 This is a problem that has, if any, a finite domain model.

 There are no function symbols.

 This is a problem that contains sort information.

 Axiom clauses: 12 Conjecture clauses: 0

 Inferences: IORe=1

 Reductions: RFMRR=2 RObv=1 RTaut=1 RFSub=1 RBSub=1 RCon=1

 Extras     : No Input Saturation, No Selection, No Splitting, Full
     Reduction,
         Ratio: 5, FuncWeight: 1, VarWeight: 1

 Precedence: div > id > app > p > new0001 > new0002 > new0003 > ind_1 >
```

```
        new0004 > ind_2 > new0005 > wind_1p > wind_2p > 0 > ls > start > T
 Ordering   : KBO
Processed Problem:


Worked Off Clauses:
3[0:Inp] || new0004(U) -> new0004(app(V,U))*.
6[0:Inp] || new0005(U) -> new0005(app(V,U))*.
11[0:Res:10.0,5.0] ||  -> new0005(0) new0004(0)*.
10[0:Res:9.1,8.0] ||  -> new0003(0)* new0005(0).
9[0:Res:1.0,2.0] ||  -> new0003(0) new0002(0)*.
2[0:Inp] || new0001(U)* -> new0003(U) new0002(U).
5[0:Inp] || new0003(U)* -> new0004(U).
8[0:Inp] || new0002(U)* -> new0005(U).
4[0:Inp] || new0004(U) -> p(U)*.
7[0:Inp] || new0005(U) -> p(U)*.
1[0:Inp] ||  -> new0001(0)*.
Usable Clauses:
52[0:Inp] || p(U)* new0002(U) -> wind_2p(U).
29[0:Inp] || p(U)* new0003(U) -> wind_1p(U).
55[0:Inp] || new0005(U) p(U)* new0002(U) -> .
54[0:Inp] || new0004(U) p(U)* new0002(U) -> .
32[0:Inp] || new0005(U) p(U)* new0003(U) -> .
31[0:Inp] || new0004(U) p(U)* new0003(U) -> .
53[0:Inp] || wind_2p(U) p(app(ind_2,U))* -> wind_2p(app(ind_2,U)).
30[0:Inp] || wind_1p(U) p(app(ind_1,U))* -> wind_1p(app(ind_1,U)).
57[0:Inp] || wind_2p(U) new0005(app(ind_2,U)) p(app(ind_2,U))* -> .
56[0:Inp] || wind_2p(U) new0004(app(ind_2,U)) p(app(ind_2,U))* -> .
34[0:Inp] || wind_1p(U) new0005(app(ind_1,U)) p(app(ind_1,U))* -> .
33[0:Inp] || wind_1p(U) new0004(app(ind_1,U)) p(app(ind_1,U))* -> .
```

```
        Given clause: 52[0:Inp] || p(U)* new0002(U) -> wind_2p(U).
        Given clause: 59[0:Res:7.1,52.0] || new0002(U)* -> wind_2p(U).
        Given clause: 60[0:Res:9.1,59.0] || -> new0003(0)* wind_2p(0).
        Given clause: 61[0:Res:60.0,5.0] || -> wind_2p(0) new0004(0)*.
        Given clause: 29[0:Inp] || p(U)* new0003(U) -> wind_1p(U).
        Given clause: 62[0:Res:4.1,29.0] || new0003(U)* -> wind_1p(U).
        Given clause: 64[0:Res:60.0,62.0] || -> wind_2p(0) wind_1p(0)*.
        Given clause: 65[0:Res:10.0,62.0] || -> new0005(0)* wind_1p(0).
        Given clause: 55[0:Inp] || new0005(U) p(U)* new0002(U) -> .
        Given clause: 32[0:Inp] || new0005(U) p(U)* new0003(U) -> .
        Given clause: 67[0:Res:7.1,55.1] || new0002(U)* -> .
        Given clause: 70[0:Res:9.1,67.0] || -> new0003(0)*.
        Given clause: 71[0:Res:70.0,62.0] || -> wind_1p(0)*.
        Given clause: 72[0:Res:70.0,5.0] || -> new0004(0)*.
        Given clause: 31[0:Inp] || new0004(U) p(U)* new0003(U) -> .
        Given clause: 73[0:Res:4.1,31.1] || new0003(U)* -> .
========== derived by MRR: 6 ==========
========== derived clauses: 18 ==========
========== subsumed clauses: 23 ==========



==== CTL-RP version 00.10 (alpha)
==== Lan Zhang, Ullrich Hustadt and Clare Dixon
==== University of Liverpool
========== Result: Valid
========== Time consumed by input:      0:00:00.0144
========== Time consumed by deduction: 0:00:00.0069
========== Derived clauses in the main loop: 27
```

```
========== Subsumed clauses in the main loop: 29
========== Successful ERES applications: 2
==========                derived clauses: 28
==========               subsumed clauses: 8
========== Failed ERES applications: 0
==========                derived clauses: 0
==========               subsumed clauses: 0
========== Total derived clauses: 55
========== Total subsumed clauses: 37
```

Bibliography

[AAB+04] Arshad Ali, Ashiq Anjum, Julian Bunn, Richard Cavanaugh, Frank van Lingen, Richard McClatchey, Harvey Newman, Wahas ur Rehman, Conrad Steenberg, Michael Thomas, and Ian Willers. Job monitoring in an interactive grid analysis environment. In *Computing in High Energy Physics*. Interlaken, Switzerland, 2004.

[ACD+08] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Peter Kilpatrick, Patrizio Dazzi, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons in gcm: Autonomic management of grid components. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:54–63, 2008. ISSN 1066-6192. URL `http://dx.doi.org/10.1109/PDP.2008.46`.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11): 56–61, 2002. ISSN 0001-0782.

[All97] Robert John Allen. *A formal approach to software architecture*. PhD thesis, Pittsburgh, PA, USA, 1997. Chair-Garlan, David.

[And94] Henrik Reif Andersen. On model checking infinite-state systems. In *In Nerode and Matiyasevich, editors, LFCS'94: Logic at St. Petersburg. Symposium on Logical Foundations of Computer Science*, pages 11–14. Springer-Verlag, 1994.

[BB04] A. Bolotov and A. Basukoski. Clausal Resolution for Extended Computation

Tree Logic ECTL+. In *Proceedings of the Time-2004/International Conference on Temporal Logic*. IEEE, Normandie, July 2004.

[BB05] A. Bolotov and A. Basukoski. Search strategies for resolution in ctl-type logics: Extension and complexity. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, (TIME 2005)*, pages 195–197. IEEE Computer Society, 2005.

[BB06] A. Basukoski and A. Bolotov. *A Clausal Resolution Method for Branching Time Logic ECTL+*, volume 46(3), chapter Annals of Mathematics and Artificial Intelligence, pages 235–263. Springer, 2006.

[BB07] A. Basso and A. Bolotov. Towards gcm re-configuration - extending specification by norms. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, 2007.

[BBB+06] A. Basso, A. Bolotov, A. Basukoski, V. Getov, L. Henrio, and M. Urbanski. Specification and verification of reconfiguration protocols in grid component systems. In *Proceedings of the 3rd IEEE International Conference on Intelligent Systems (IS-2006)*, pages 450–455. IEEE, Los Alamitos, USA, 2006.

[BBC05] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74(1):45–54, 2005. ISSN 0164-1212.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer-Verlag, January 2006. URL `http://www-sop.inria.fr/oasis/proactive/userfiles/file/papers/ProgrammingComposingDeploying.pdf`.

[BBG09] Alessandro Basso, Alexander Bolotov, and Oleg Grigoriev. Deontic extension of deductive verification of component model: Combining computation tree logic and

deontic logic in natural deduction style calculus. In *International Indial Conference on Artificial Intelligence*, 2009.

[BBGH08] A. Basso, A. Bolotov, V. Getov, and L. Henrio. Dynamic reconfiguration of gcm components. Technical Report TR-0173, CoreGRID, 2008.

[BBGS05] Alexander Bolotov, Vyacheslav Bocharov, Alexander Gorchakov, and Vasilyi Shangin. Automated first order natural deduction. In *IICAI*, pages 1292–1311, 2005.

[BBGS06] A. Bolotov, A. Basukoski, O. Grigoriev, and V. Shangin. Natural deduction calculus for linear-time temporal logic. In *Joint European Conference on Artificial Intelligence (JELIA-2006)*, pages 56–68, 2006.

[BCD+09] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1):5–24, 02 01, 2009. URL `http://dx.doi.org/10.1007/s12243-008-0068-8`.

[BCF02] A. Bolotov, C.Dixon, and M. Fisher. *On the Relationship between Normal Form and w-automata (with M.Fisher and C.Dixon)*, volume 12, pages 561–581. Oxford University Press, 2002.

[BCH+02] Françoise Baude, Denis Caromel, Fabrice Huet, Lionel Mestre, and Julien Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 93. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1686-6.

[BCS02] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Seventh Int. Workshop on Component-Oriented Programming (WCOP02), at ECOOP*, 2002.

[BCS04] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model. 2004. URL `http://fractal.objectweb.org/specification/fractal-specification.pdf`.

[BD00] A. Bolotov and C. Dixon. Resolution for Branching Time Temporal Logics: Applying the Temporal Resolution Rule. In *Proceedings of the 7th International Conference on Temporal Representation and Reasoning (TIME2000)*, pages 163–172. IEEE Computer Society, Cape Breton, Nova Scotia, Canada, 2000.

[BF99] A. Bolotov and M. Fisher. A Clausal Resolution Method for CTL Branching Time Temporal Logic. *Journal of Experimental and Theoretical Artificial Intelligence.*, 11:77–93, 1999.

[BFH03] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 0470853190.

[BG08] A. Bolotov and O. Grigoriev. Combining computation tree logic and deontic logic in natural deduction style calculus. Technical report, 2008.

[BGS06] A. Bolotov, O. Grigoriev, and V. Shangin. Natural deduction calculus for computation tree logic. In *IEEE John Vincent Atanasoff Symposium on Modern Computing*, pages 175–183, 2006.

[BGS07] Alexander Bolotov, Oleg Grigoriev, and Vasilyi Shangin. Automated natural deduction for propositional linear-time temporal logic. In *TIME '07: Proceedings of the 14th International Symposium on Temporal Representation and Reasoning*, pages 47–58. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2836-8.

[BGTI08] A. Basukoski, V. Getov, J. Thiyagalingam, and S. Isaiadis. *Component-Based Development Environment for Grid Systems: Design and Implementation*, chapter

Making Grids Work, Institute of Computer Science, Foundation for Research and Technology, pages 119–128. Springer, Hellas in Crete, Greece, 2008.

[BH95] Jonathan P. Bowen and Michael G. Hinchey, editors. *Applications of Formal Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995. ISBN 0133669491.

[BHC+06] T. Barros, L. Henrio, A. Cansado, E. Madelaine, M. Moreland V. Mencl, and F. Plasil. Extension of the fractal adl for the specification of behaviours of distributed components. In *Accepted for poster presentation at the 5th Fractal Workshop (part of ECOOP'06)*, 2006.

[BHM05] T. Barros, L. Henrio, and E. Madelaine. Verification of distributed hierarchical components. In *Proc. of the International Workshop on Formal Aspects of Component Software (FACS'05)*, pages 41–55. Electronic Notes in Theor. Computer Sci (ENTCS), 2005.

[BJ08] LT. Bao and M. Jones. Refinement for predicate abstraction in the context of abstract component model. Technical Report TR SMC-BYU-0109, Dept. of Computer Science, Brigham Young University, Provo, UT, 2008.

[BJC05] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *Software Architecture*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2005. URL `http://dx.doi.org/10.1007/11494713_1`.

[BMV96] David Basin, Sean Matthews, and Luca Vigano. Natural deduction for non-classical logics, 1996.

[Bol00] A. Bolotov. *Clausal Resolution for Branching-Time Temporal Logic.* PhD thesis, Department of Computing and Mathematics, The Manchester Metropolitan University, 2000.

[Bol03] A. Bolotov. Clausal resolution for extended computation tree logic ectl. In *Proceedings of the Time-2003/International Conference on Temporal Logic 2003*. IEEE, 2003.

[Bol05] A. Bolotov. Autogrid: Temporal modelling of intelligent grids. CoreGRID WP3 meeting, 2005. URL `http://www.di.unipi.it/~marcod/WP3homepage/JuneMeeting/Contrib/AlexanderBolotov.pdf`.

[Box98] Don Box. *Essential COM*. Addison-Wesley, 1998.

[BR00] Thais Batista and Noemi Rodriguez. Dynamic reconfiguration of component-based applications. *Software Engineering for Parallel and Distributed Systems, International Symposium on*, 0:32, 2000. ISBN 0-7695-0634-8.

[BS03] P. Bidinger and J. Stefani. The kell calculus: operational semantics and type system. 2003.

[CC99] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engg.*, 6(1):69–95, 1999. ISSN 0928-8910.

[CCH+08] Antonio Cansado, Denis Caromel, Ludovic Henrio, Eric Madelaine, Marcela Rivera, and Emil Salageanu. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, chapter A Specification Language for Distributed Components implemented in GCM/ProActive. Springer, 2008. `http://agrausch.informatik.uni-kl.de/CoCoME`.

[CDG01] Marsha Chechik, Benet Devereux, and Arie Gurfinkel. Model-checking infinite state-space systems with fine-grained abstractions using spin. In *In Proceedings of the 8th SPIN Workshop on Model Checking Software, volume 2057 of LNCS*, pages 16–36. Springer, 2001.

[CFJV05] E. M. Clarke, A. Fehnker, S. Jha, and H. Veith. *Temporal Logic Model Checking*, volume Handbook of Networked and Embedded Control Systems of *Control Engineering*, chapter IV, pages 539–558. Birkhauser Boston, 1 edition, 2005.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. ISSN 0164-0925.

[CIY95] Rance Cleaveland, S. Purushothaman Iyer, and Daniel Yankelevich. Optimality in abstractions of model checking. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 51–63. Springer-Verlag, London, UK, 1995. ISBN 3-540-60360-3.

[CJM98] E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *In Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[Cla97] Edmund Clarke. Model checking. *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, 1997. URL `http://dx.doi.org/10.1007/BFb0058022`.

[Cor] Microsoft Corp. Introduction to activex controls. URL `http://microsoft.com/`.

[CR94] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In *CONCUR'94*. LNCS 836, Springer, 1994. ISBN 3-540-58329-7.

[Cro97] J. Crow, editor. *NASA Formal Methods Guidebook Vol. 2*, volume 2 of *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems.* NASA-GB-001-97, 1997.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996. ISSN 0360-0300.

[CXDM04] S. Caballe, F. Xhafa, T. Daradoumis, and J.M. Marques. Towards a generic platform for developing cscl applications using grid infrastructure. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:200–207, 2004. ISBN 0-7803-8430-X.

[DD07] CoreGRID Deliverable D.PM.04. Basic features of the grid component model. March 2007.

[DFB02] C. Dixon, M. Fisher, and A. Bolotov. Clausal Resolution in a Logic of Rational Agency. *Artificial Intelligence*, 2002.

[DFK07] C. Dixon, M. Fisher, and B. Konev. Tractable temporal reasoning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 318–323, 2007.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997. ISSN 0164-0925.

[DJ93] J.V. D'Anniballe and P.J. Koopman Jr. Towards execution models of distributed systems: A case study of elevator design. In *International Workshop on Hardware-Software Co-Design*, page 9, 1993.

[EDD+04] Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors. *Integration of Software Specification Techniques for Applications in Engineering, Priority Program Soft-Spez of the German Research Foundation (DFG), Final Report*, volume 3147 of *Lecture Notes in Computer Science*. Springer, 2004. ISBN 3-540-23135-8.

[EG92] Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artif. Intell.*, 57(2-3):227–270, 1992. ISSN 0004-3702.

[Eme90] E. A. Emerson. *Temporal and Modal Logic*, pages 996–1072. Elsevier, 1990.

[Eme08] E. Allen Emerson. The beginning of model checking: A personal perspective. *Lecture Notes in Computer Science*, 5000:27–45, 2008. ISBN 978-3-540-69849-4.

[faEIG] GridComp An Advanced Component Platform for an Effective Invisible Grid. The gridcomp ibm use case. URL `http://gridcomp.ercim.org/content/view/41/39/#IBM`.

[FKNT02] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. 2002.

[Fon93] Leonard N. Foner. What's an agent anyway? - a sociological case study. FTP Report - MIT Media Lab, May 1993.

[Fou09] Eclipse Foundation. Eclipse open development platform, November 2009. URL `http://www.eclipse.org/`.

[Fra95] Frederick K. Frantz. A taxonomy of model abstraction techniques. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, pages 1413–1420. IEEE Computer Society, Washington, DC, USA, 1995. ISBN 0-7803-3018-8.

[FT05] Ian Foster and Steven Tuecke. Describing the elephant: The different faces of it as service. *Queue*, 3(6):26–29, 2005. ISSN 1542-7730. URL `http://dx.doi.org/10.1145/1080862.1080874`.

[GBT+07] V. Getov, A. Basukoski, J. Thiyagalingam, Y. Yulai, and Y. Wu. Grid programming with components : an advanced component platform for an effective invisible grid. Technical report, GRIDComp Technical Report, 2007.

[Gen35] G. Gentzen. Untersuchungen uber das logische schliessen. *Mathematische*

Zeitschrift, 39:176–210, 1935. English translation in The Collected Papers of Gerhard Gentzen, M.E. Szabo (Ed.), North-Holland, Amsterdam, 1969.

[GRSF04] C. Goble, D. De Roure, N.R. Shadbolt, and A.A.A. Fernandes. Enhancing services and applications with knowledge and semantics. *The Grid 2: Blueprint for a New Computing Infrastructure*, pages 431–458, 2004.

[Hig00] James A. Highsmith. *Adaptive software development: a collaborative approach to managing complex systems*. Dorset House Publishing Co., Inc., New York, NY, USA, 2000. ISBN 0-932633-40-4. URL `http://portal.acm.org/citation.cfm?id=323922`.

[HL98] Y.-W. Hsieh and S. P. Levitan. Model abstraction for formal verification. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 140–147. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8359-7.

[HT87] T. Hafer and W. Thomas. Computation tree logic ctl* and path quantifiers in the monadic theory of the binary tree. *ICALP*, 267:269–279, 1987.

[JCK98] Task Lead John C. Kelly, editor. *NASA Formal Methods Guidebook Vol. 1*, volume Volume I: Planning and Technology Insertion. NASA/TP-98-208193, 1998.

[Kel94] P. Kelb. Model checking and abstraction: A framework approximating both truth and failure information. Technical report, University of Oldenburg, 1994.

[KF98] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 01, 1998. ISBN 1558604758. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1558604758`.

[KMWM03] Richard Krutisch, Philipp Meier, Martin Wirsing, and Ludwig Maximilians. The agent component approach, combining agents and components. In *In Proceedings*

of MATES-03, Springer series of Lecture Notes on Artificial Intelligence. Universtitat Munchen, 2003.

[KPP+04] Mikhail F. Kanevski, Roman Parkin, Aleksey Pozdnukhov, Vadim Timonin, Michel Maignan, Vasiliy V. Demyanov, and Stéphane Canu. Environmental data mining and modeling based on machine learning algorithms and geostatistics. *Environmental Modelling and Software*, 19(9):845–855, 2004.

[Lin01] Jürgen Lind. Relating agent technology and component models, 2001.

[LW06] A. Lomuscio and B. Wozna. *A complete and decidable axiomatisation for deontic interpreted systems*, volume 4048, pages 238–254. Springer, 2006.

[Mar94] Assaf Marron. Method of operating a data processing system having a dynamic software update facility. (5359730), October 1994. URL `http://www.freepatentsonline.com/5359730.html`.

[Mci68] D. Mcilroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[MHW03] Richard Monson-Haefel and A. K. Weissinger. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003. ISBN 059600530X.

[MMHR04] J. Matevska-Meyer, W. Hasselbring, and R.H. Reussner. Software architecture description supporting component deployment and system runtime reconfiguration. In *Proceedings of the Ninth International Workshop on Component-Oriented Programming*, 2004.

[MP92] Z. Manna and A. Pnueli. Temporal specification and verification of reactive modules. Technical report, Weizmann Institute of Science, 1992.

[oE] CoreGRID Network of Excellence. Coregrid - the european research network on foundations, software infrastructures and applications for large scale distributed, grid and peer-to-peer technologies. URL `http://www.coregrid.net/`.

[OMG06] Object Management Group OMG. Corba component model specification, 2006.

[Pfe01] Frank Pfenning. Logical frameworks. pages 1063–1147, 2001. ISBN 0-444-50812-0.

[PFT03] Mónica Pinto, Lidia Fuentes, and Jose María Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 118–137. Springer-Verlag New York, Inc., New York, NY, USA, 2003. ISBN 3-540-20102-5.

[PPK06] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 0:133–141, 2006. ISSN 1550-6215.

[Pro08] The Grid5000 Project. An infrastructure distributed in 9 sites around france, for research in large-scale parallel and distributed systems, 2008. URL `http://www.grid5000.fr`.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002. ISSN 0098-5589.

[RBBM04] Tomas Barros Rabea, Tomás Barros, Rabéa Boulifa, and Eric Madelaine. Parameterized models for distributed java objects. In *In Forte'04 conference, Madrid, 2004. LNCS 3235, Spinger Verlag*, pages 43–60. Spinger Verlag, 2004.

[Rus02] John C. Russ. *Image Processing Handbook, Fourth Edition.* CRC Press, Inc., Boca Raton, FL, USA, 2002. ISBN 084931142X.

[SBS09] Ali Selamat, Siti Dianah Bujang, and Md. Hafiz Selamat. Agent verification design of short text messaging system using formal method. In *KES-AMSTA '09: Proceedings of the Third KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, pages 514–522. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-01664-6.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software – Beyond Object-Oriented Programming – Second Edition*. Addison-Wesley and ACM Press, 2002. ISBN 0-201-74572-0.

[She94] Wei-Min Shen. *Autonomous Learning from the Environment*. W. H. Freeman & Co., New York, NY, USA, 1994. ISBN 0716782650.

[Sim94] Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994. URL `http://homepages.inf.ed.ac.uk/als/Research/thesis.ps.gz`.

[SK04] E. A. Strunk and J.C. Knight. Assured reconfiguration of embedded real-time software. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 367. IEEE Computer Society, 2004.

[Ste99] Benno Stein. Generating heuristics to control configuration processes. *Applied Intelligence*, 10(2-3):247–255, 1999. ISSN 0924-669X.

[Sto07] Heinz Stockinger. Defining the grid: a snapshot on the current view. *J. Supercomput.*, 42(1):3–17, October 2007. ISSN 0920-8542. URL `http://dx.doi.org/10.1007/s11227-006-0037-9`.

[Tal02] Domenico Talia. The open grid services architecture: Where the grid meets the web. *IEEE Internet Computing*, 6(6):67–71, 2002. ISSN 1089-7801.

[TIG04] J. Thiyagalingam, S. Isaiadis, and V. Getov. *Towards Building a Generic Services Platform: A Components-Oriented Approach*. Springer-Verlag, 2004.

[TL02] Peter A. Tanner and Po-Tak Law. Effects of synoptic weather systems upon the air quality in an asian megacity. *Water, Air, & Soil Pollution*, 136(1):105–124, 05 01, 2002. URL `http://dx.doi.org/10.1023/A:1015275404592`.

[VAHL02] Mauricio Varea, Bashir Al-Hashimi, and Michael Leuschel. Finite and infinite model checking of dual transition petri net models (extended abstract). In *Second Workshop on Automated Verification of Critical Systems (AVOCS)*, pages 265–269. Birmingham, UK, 2002.

[VGK04] Sergiy A. Vilkomir, Aditya K. Ghose, and Aneesh Krishna. Combining agent-oriented conceptual modelling with formal methods. *Australian Software Engineering Conference*, 0:147, 2004. ISSN 1530-0803.

[WADG09] Thomas Weigold, Marco Aldinucci, Marco Danelutto, and Vladimir Getov. Integrating autonomic grid components and process-driven business applications. In *Proc of Autonomics: 3rd Intl. ICST Conference on Autonomic Computing and Communication Systems*. Springer, 2009.

[WK02] Martin Wirsing and Alexander Knapp. A formal approach to object-oriented software engineering. *Theor. Comput. Sci.*, 285(2):519–560, 2002. ISSN 0304-3975.

[Wol85] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, (110–111):119–136, 1985.

[Wol95] P. Wolper. On the relation of programs and computations to models of temporal logic. In Leonard Bolc and Andrzej Szałas, editors, *Time and Logic, a computational approach*, chapter 3, pages 131–178. UCL Press Limited, 1995.

[Wol01] Pierre Wolper. Constructing automata from temporal logic formulas: A tutorial. *Lectures on Formal Methods and PerformanceAnalysis*, pages 261–277, 2001. URL `http://dx.doi.org/10.1007/3-540-44667-2_7`.

[Woo00] M. Wooldridge. *Reasoning about Rational Agents.* MIT Press, July 2000.

[WS01] R. Weinreich and J. Sametinger. *Component-based software engineering: putting the pieces together*, chapter Component models and component services: concepts and principles, pages 33–48. Addison-Wesley Longman Publishing Co., Inc., 2001.

[XZ08] Zhi Bin Xue and Jian Chao Zeng. A novel exponential type swarming of foraging and obstacle-avoidance behaviour modelling and simulating research on collective motion in multi-obstacle environment. In *ISICA '08: Proceedings of the 3rd International Symposium on Advances in Computation and Intelligence*, pages 454–460. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-92136-3.

[ZHD08] L. Zhang, U. Hustadt, and C. Dixon. First-order resolution for ctl. Technical report, ULCS-Department of Computer Science, University of Liverpool, 2008. 08–010 pp.

[Zit07] Matthew Zito. The many flavors of grid computing, 2007. URL `http://www.gridapp.com/resources/pdf/nov27webinar.pdf`.

Index