UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

École Doctorale STIC
Sciences et Technologies de l'Information et de la Communication

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Mention Informatique

présentée et soutenu par

Muhammad Uzair KHAN

A Study of First Class Futures: Specification, Formalisation, and Mechanised Proofs

Thèse dirigée par Ludovic HENRIO et Denis CAROMEL

au sein de l'équipe OASIS,

*équipe commune de l'*INRIA Sophia Antipolis*, du* CNRS *et du laboratoire* I3S

soutenue le 25 Février 2011,devant le jury composé de:

Président du Jury	Yves Bertot	INRIA-Sophia Antipolis
Rapporteurs	Jean-Bernard Stefani	INRIA Grenoble-Rhône-Alpes, France
	Christian Perez	INRIA-ENS Lyon, France
	Carlos Canal	ETSI Informática Universidad de Málaga, Espagne
Directeur de thèse	Denis Caromel	INRIA-CNRS-Université de Nice Sophia Antipolis
Co-directeur	Ludovic Henrio	INRIA-CNRS-I3S–Sophia Antipolis

0.1 Acknowledgment

Last thing to do :-)

Contents

0.1 Acknowledgment . . . i

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Results and Contributions . . . 3
1.2.1 Specification and Implementation of Future Update Strategies 4
1.2.2 Formalisation of Component Model and Proofs . . . 4
1.3 Impact of Thesis . . . 5
1.4 Thesis Outline . . . 5

2 Related Works **9**
2.1 Background: Why Futures ? . . . 10
2.1.1 Some basic questions about futures . . . 11
2.2 Distributed and Concurrent Programming . . . 13
2.3 Distributed Concurrent Programming with Futures . . . 14
2.3.1 Futures in Multilisp World . . . 14
2.3.2 Futures in ABCL/1 and ABCL/f . . . 17
2.3.3 Futures in Alice ML and $\lambda(fut)$ . . . 19
2.4 Distributed Concurrent Programming with Futures and Objects . . . 20
2.4.1 Future in Java-verse . . . 20
2.4.2 Futures in Creol . . . 22
2.4.3 Futures in ASP, ProActive and ASP_{fun} . . . 24
2.4.4 Futures in AmbientTalk . . . 25
2.5 Component Models and Frameworks . . . 27
2.5.1 Common Object Model (COM) and DCOM . . . 28
2.5.2 Enterprise Java Beans . . . 28
2.5.3 CORBA Component Model (CCM) and GridCCM . . . 29
2.5.4 Common Component Architecture (CCA) . . . 30
2.5.5 Service Component Architecture (SCA) and FraSCAti . . . 31
2.5.6 SOFtware Appliances Component Model (SOFA) . . . 32
2.5.7 Fractal component model . . . 33
2.5.8 Grid Component Model (GCM) . . . 34
2.6 Summary of Related Works and Positioning . . . 36

I Future Update Strategies: Specification and Implementation **41**

3 First Class Futures: Specification of Update Strategies **43**
3.1 Background: Futures in ASP-Calculus . . . 45
3.2 Background: Update Strategies for Futures . . . 47
3.2.1 Classification of Future Update strategies . . . 47

3.2.2 Eager Forward-based Strategy 48
3.2.3 Eager Message-based Strategy 49
3.2.4 Lazy Message-based Strategy 51
3.3 Semi-Formal Specification of Update Strategies 52
3.3.1 General Notation 52
3.3.2 Eager Forward-based Strategy 55
3.3.3 Eager Message-based Strategy 56
3.3.4 Lazy Message-based Strategy 58
3.4 Analysis of Future Update Strategies 59
3.5 Remarks on Semi-formal Specification of Strategies 62

4 Implementing Future Update Strategies in ProActive 65
4.1 Background: First Class Futures in ProActive 66
4.1.1 First Class Futures in ProActive: Automatic Continuation . . 68
4.2 Missing Future Update Strategies 70
4.2.1 Eager Message-based Strategy 70
4.2.2 Lazy Message-based Strategy 72
4.3 Experimental Evaluation 74
4.4 Concluding Remarks on Future Update Strategies 79

II Formal Reasoning on Components: Semantics and Proofs 81

5 A Framework for Reasoning on Component Composition 83
5.1 Background: Isabelle/HOL 85
5.1.1 Isabelle/HOL Syntax 86
5.2 An Asynchronous Component Model with Futures 89
5.2.1 Component Model Overview 90
5.2.2 Component Structure 90
5.2.3 Communication Model 92
5.2.4 Component Behaviour 93
5.2.5 Why First Class Futures in GCM ? 94
5.3 Formalisation of a Component Model in Isabelle/HOL 96
5.3.1 Component Structure 97
5.3.2 Efficient Specification of Component Manipulation 98
5.3.3 Component State 103
5.3.4 Correct Component 106
5.3.5 Basic Properties on Component Structure and Manipulation . 107
5.3.6 Properties on Component Correctness 109
5.4 Runtime Reconfiguration of Components 111
5.4.1 Complete Component 112
5.4.2 Reconfiguration Primitives: Unbind and Replace 113

6 Asynchronous Components with Futures : Semantics and Proofs 117
6.1 An Asynchronous Component Model with Futures 119
6.2 Run time Semantics for GCM-like Components 121
6.2.1 Structure and Notations 121
6.2.2 Semantics of Component Model 126
6.3 Formalisation in Isabelle and Properties 133
6.3.1 Semantics 134
6.3.2 Properties and Proofs on Eager message-based Strategy . . . 135

7 Positioning and Concluding Remarks on Formalisation 139

8 Conclusion 143
8.1 Final remarks 150

9 Future Works 153
9.1 Applied Aspects 153
9.2 Theoretical Aspects 157

A Summary of terms and notations 161

B Semantics of Lazy message-based Strategy 167

Bibliography 171

List of Figures

3.1 Futures propagate throughout the system . . . 46
3.2 Eager forward-based: Future updates follow the flow of futures . . . 48
3.3 Eager message-based: All future recipients register . . . 50
3.4 Lazy message-based: Register only on wait-by-necessity . . . 51
3.5 Future-update in eager forward-based strategy . . . 56
3.6 Future-update in eager message-based strategy . . . 57
3.7 Future update in lazy message-based strategy . . . 58

4.1 Anatomy of an Active Object . . . 67
4.2 Active objects and futures in ProActive . . . 68
4.3 A small example tree configuration . . . 75
4.4 Comparison of strategies for a tree configuration . . . 76
4.5 Pipe of varying length . . . 77
4.6 Comparison of strategies for a pipe configuration . . . 78

5.1 High level view of a GCM component [1] . . . 90
5.2 Component composition . . . 91
5.3 Structure of a primitive component . . . 92
5.4 Example composite component . . . 92
5.5 Behaviour of primitive components . . . 94
5.6 First Class Futures in GCM (a) . . . 94
5.7 First Class Futures in GCM (b) . . . 95
5.8 First Class Futures in GCM (c) . . . 95
5.9 First Class Futures in GCM (d) . . . 96
5.10 Composite Component . . . 98

6.1 Future registration . . . 120
6.2 Future update . . . 121
6.3 Structure and behaviour of a primitive component . . . 123
6.4 Primitive Component Semantics . . . 127
6.5 Component Communications . . . 129
6.6 CompositeCall . . . 129
6.7 Semantics of the component composition (a) . . . 130
6.8 CommBrother . . . 131
6.9 CommChild rule . . . 132
6.10 CommParent . . . 132
6.11 Semantics of the component composition (b) . . . 133

B.1 Primitive Component Semantics (Lazy message-based) . . . 167
B.2 Semantics of the component composition (a) . . . 168
B.3 Semantics of the components . . . 169

Chapter 1

Introduction

Contents

1.1 Motivation **2**
1.2 Results and Contributions **3**
1.2.1 Specification and Implementation of Future Update Strategies 4
1.2.2 Formalisation of Component Model and Proofs 4
1.3 Impact of Thesis **5**
1.4 Thesis Outline **5**

1.1 Motivation

Ever since the introduction and first serious applications of computer networks in 1970-1980s, the field of distributed computing has been growing enormously. Starting from their humble roots as E-mail application of ARPANET and the Usenet discussion system, distributed systems have become prevalent in modern life. Every aspect of modern life is somehow influenced by one or more applications of distributed computing. At the same time, the computational capabilities of individual microprocessors have grown exponentially. Today's processors used in standalone machines are more powerful than some of the earliest mainframes. To better harness the available processing power, and to optimise the computational efficiency, a lot of effort has been put on studying concurrency, parallelism and distribution.

Distributed systems have been described in literature in a number of ways, depending upon factors such as distribution model, communication model, etc. We view a distributed system as a set of *concurrent communicating processes*. Such a view allows us to benefit naturally from distribution and is somewhat close to the *Actor Model* [2] for concurrent computation. Traditionally, concurrency is achieved by creating separate tasks/processes or threads that execute some part of the code in a concurrent manner. While such constructs do allow the programmers to specify which sections of the code may be executed in parallel, they are very intrusive to the business logic. Problems abound relating to concurrency and special attention has to be paid on access to shared resources and synchronisation between the various concurrently executing tasks/threads/processes to ensure that the behaviour of the concurrent/parallelised program stays *correct* with respect to the specification.

Futures were introduced to simplify creation of concurrent applications. For concurrent distributed applications, futures enable an efficient and easy to use programming paradigm. A future is a placeholder for a result of a concurrent execution. When the concurrent execution finishes, the produced result replaces the placeholder object. First introduced in Multilisp [3] and ABCL/1 [4], futures provide an easy notation style construct for converting sequential code to concurrent code, abstracting away complexities of concurrency management. Although not traditionally a part of popular industrial use programming languages, futures appear widely in literature. A number of languages and formalisations have been proposed which benefit from futures. These range from Multilisp, ABCL/f, λ-(fut) [5] to Creol [6], ASP-calculus [7], and AmbientTalk [8], etc. Java supports the future construct through `java.util.concurrent` package.

Futures can be treated as *first class* objects. First class futures act as normal objects and may be passed as method arguments. Similarly, return values from method calls may also contain futures. First class futures allow us to manipulate (to some extent) the yet-to-compute results of concurrent executions. In a distributed application, first class futures can be transmitted between the communicating remote processes. Consequently, futures may spread everywhere. This is indeed the case for ASP-calculus for example, which allows first class futures. When the concurrent execution corresponding to a future finishes, the produced result has to be

communicated to all processes/objects which have received that future. Additional mechanisms are implemented to facilitate transmitting the produced result values to where they are needed. We refer to those mechanisms as *future update strategies*.

First class futures and by extension future update strategies are somewhat neglected in literature. Even in case of works such as Creol, not much attention is paid to the role of future update strategies. On the other hand, ASP-calculus explicitly discusses different future update strategies; those strategies can be applied to other frameworks with first class futures as well, such as Creol. Under reasonable hypothesis it has been shown that the order in which results are returned has no influence on the computation. This opens the possibility of improving the efficiency of result transmission under different application and network configurations. Consequently, **this thesis focuses on the study and formalisation of different future update strategies**.

1.2 Results and Contributions

The work presented in this thesis can be split into two broad parts. In Part I, we study the transmission of results for first class futures. We present the protocols for future update strategies in detail, and provide a semi-formal specification of those update strategies. Our presented specification is generic and language or framework independent; it can be adapted to other frameworks/languages with first class futures. Mainly, in Part I we focus on:

- a semi-formal specification of future update strategies.
- an implementation of future update strategies in ProActive.

Part II of this thesis deals with more theoretical aspects. We formalise a component model with asynchronous communications, and futures. Our intent is to build an infrastructure which is sufficient for proving properties on component models in a theorem prover. We aim for an expressive platform with a wide enough range of tools and support lemmas to help design of component models, the creation of adaptive procedures, and proof of generic properties on component models. Using the developed infrastructure, we present formal runtime semantics of our components. Our component semantics include formalisation of future update strategies. It provides a reliable and strong basis for reasoning on components, and on future update protocols. Our work serves to prove the correctness of the underlying middleware implementation. To summarise, in Part II we:

- formalise a component model and mechanically prove the correctness of properties on our formalised model.
- give a runtime semantics for our components including the specification of a future update strategy. We prove properties on correctness of future update mechanisms.

1.2.1 Specification and Implementation of Future Update Strategies

Future update strategies are somewhat neglected in literature and to our knowledge, ours is the only work that focuses on future updates. A high level specification of update strategies appear in [7]. However, the specification is abstract, and lacks details on working of the various strategies or their underlying data structures. Consequently, it is not very useful for insight into a real implementation of update strategies. We present a semi-formal specification which is more precise and is detailed enough to be used as basis for a real implementation. Additionally, our specification can also be used to analyse the efficiency of strategies. Our semi-formal specification is generic and language independent. Consequently, outside ASP and ProActive, this work could be adapted to other frameworks that use first class futures, for example, Creol and AmbientTalk.

We aim to evaluate and to study the efficiency of the specified future update strategies. Based on our semi-formal specification, we provide an implementation of those strategies in ProActive distributed programming library. We carry out some experiments to measure the efficiency of future update strategies under different configurations. Our goal is to show that *no single strategy provides optimal performance for all situations*. Having multiple strategies allows the selection of the most suited strategy for a particular application configuration. Although, we provide some insights on which strategy is more more suited in certain context, we do not provide a large scale experimental case study. Such a case study is out of the scope of this thesis and is left as part of future work.

Our work on semi-formal specification of future update strategies and their experimental evaluation is published in [9, 10].

1.2.2 Formalisation of Component Model and Proofs

As already stated, the theoretical part of our thesis focuses on following goals: formalisation of a component model, and reasoning on components at runtime; In particular we prove properties on the future update protocols.

We formalise a subset of GCM [11], and build a framework for reasoning on distributed components with futures using Isabelle/HOL [12] theorem prover. Component models ensure that components have a well-determined structure which facilitates reasoning on component interactions. We mechanically prove properties on correctness of components, explaining our design choices along the way. To demonstrate that our reasoning infrastructure is sufficiently detailed, we also show some initial proofs on component configuration and reconfiguration. The details on our framework for reasoning on components are published in [13].

Proving correctness of protocols is a complex task. To accomplish this goal in the special case of future update protocols, we formally specify runtime semantics of distributed components with futures, incorporating formalisation of future update strategies. Formalising future updates is of little interest concerning the language

properties, but is crucial to study the implementation of this language. In order to prove the correctness of our implementation of GCM, we formalise one future update strategy and prove correctness of future updates. We prove two main properties concerning futures. The first property ensures the correctness of future update operation and verifies that *a future update removes all references to a given future.* The second property establishes the correctness of formalised future update protocol and verifies that *given a correct component system, the global reduction maintains complete future registrations.* More details on future update strategies appear in Chapter 3, while Chapter 6 describes the proofs. Our work on runtime semantics and correctness proofs of future update protocols is published in [14].

1.3 Impact of Thesis

Concurrency and distribution are widely studied topics in the research communities. A large body of work exists dealing with various aspects of concurrency and distribution. In this thesis, we study efficient transmission of results in distributed systems with futures. As stated, there are no directly competing works on future update strategies.

We provide a semi-formal specification, which we believe is detailed enough to be used as basis for a real implementation. We give our semi-formal specification of future update strategies through a language independent approach that makes it applicable to various existing frameworks that support first class futures, for example Creol and AmbientTalk. We provide an implementation of our update strategies in ProActive which may serve as reference for supporting similar strategies in other frameworks using first class futures.

We formalise our future update strategies in the context of GCM component model. Ours is the only formalisation available for GCM, and can be used for proving correctness of its reference implementation ProActive/GCM. To the best of our knowledge, our work is the only one which focuses on efficient transmission of future values, and attempts to formalise the value-transmission mechanisms for first class futures with the goal of providing mechanised proofs on the properties of update strategies. This serves to prove correctness of the corresponding implementation of future update strategies in ProActive library [15]. We believe that our formalisation can directly benefit other similar component models as well; for example FraSCAti component model which has a lot of similarities with GCM. More details on component models and their relationship to our work appear in Chapters 2 and Chapter 7.

1.4 Thesis Outline

As stated before, our contributions are grouped in two different parts. Chapter 3 and Chapter 4 form Part I, which is concerned with specification and implementation of future update strategies. Part II is more formal and contains: Chapter 5, Chapter 6

presenting our formalism and proofs, and Chapter 7 which positions our work with respect to other works on component models and formal reasoning on components. The thesis is organised as follows:

Chapter 2. Related Works

We evaluate other works that make use of futures or similar constructs as means for providing concurrency and parallelism. We study the various ways in which the futures or similar constructs are implemented/reasoned on, and discuss how those works relate to the work presented in this thesis. We overview the more common component models and their formalisation, aimed at large scale distributed systems and computational grids. Finally, we position our work with respect to existing works on futures and distributed component models.

Chapter 3. First Class Futures: Specification of Update Strategies

We present a semi-formal event-like notation to model the future update strategies. For our specification, we build upon the high level definitions provided in [7] to present a detailed semi-formal specification of update strategies using a generalised, language independent notation. We particularly study the efficiency of each strategy, and present a basic cost model (w.r.t. number of message exchanges) to evaluate each of the presented strategies.

Chapter 4. Implementing Future Update Strategies in ProActive

We use the semi-formal specifications shown in Chapter 3 to demonstrate how such strategies may be implemented. We implement the future update strategies in ProActive, a distributed programming library based on ASP-calculus. We aim to better evaluate the various future update strategies, and to study when a particular strategy might be more suitable. We carry out initial experiments, comparing the efficiency of each strategy to validate the results and analysis of previous chapter. We only present initial experimental results, leaving a large scale and exhaustive experimental evaluation for future work.

Chapter 5. A Framework for Reasoning on Component Composition

We present the formalisation in Isabelle/HOL of a component model with futures, focusing on the structure and on basic lemmas to handle component structure. Correctness of component composition is well understood formally but existing works do not allow for mechanised reasoning on composition and component reconfiguration. A mechanical support improves the confidence in the existing results. Our objective is to present the basic constructs, and the corresponding lemmas allowing the proof of properties related to structure of component models and the handling of structure at runtime. First proofs on component configuration and reconfiguration are also presented to demonstrate the expressiveness of underlying reasoning infrastructure.

Chapter 6. Components with Futures : Semantics and Proofs

We give a formal semantics to the component model formalised in Chapter 5. The runtime semantics of our component model incorporate formalisation of one future update strategy from Chapter 3. We only show proofs on one future update strategy and present the semantics of a second strategy in Appendix B, showing that our approach is well adapted to specify and reason on different future update strategies. Our model has been mechanically formalised in Isabelle/HOL, together with the proof of properties on future update strategy. This demonstrates the correctness of update strategy and validates the actual implementation of the strategy itself.

Chapter 7. Positioning and Concluding Remarks on Formalisation

We summarise our work on formalisation of a component model with first class futures and its runtime semantics. We contrast this work with the previously existing work, and clearly distinguish between previous work and our contribution. We highlight the properties that distinguish our formalised model from GCM and its reference implementation ProActive/GCM. The goal of the chapter is to position our work with respect to previously published work work on components models and formalisations that support reasoning on them.

Chapter 8. Conclusion

We summarise the work presented in thesis; in particular we focus on the main contributions of this thesis, and their impact. We briefly discuss how our work relates to other languages/frameworks (already presented in other chapters), and provide concluding remarks for the thesis.

Chapter 9. Future Works

The work presented in the thesis is not exhaustive. Rather we hope to provide a strong basis for further research on future update protocols. We present some perspectives for future efforts, discussing some short term and long term objectives, and future directions.

Chapter 2

Related Works

Contents

2.1 Background: Why Futures ? 10

2.1.1 Some basic questions about futures 11

2.2 Distributed and Concurrent Programming 13

2.3 Distributed Concurrent Programming with Futures 14

2.3.1 Futures in Multilisp World 14

2.3.2 Futures in ABCL/1 and ABCL/f 17

2.3.3 Futures in Alice ML and $\lambda(fut)$ 19

2.4 Distributed Concurrent Programming with Futures and Objects 20

2.4.1 Future in Java-verse 20

2.4.2 Futures in Creol 22

2.4.3 Futures in ASP, ProActive and ASP_{fun} 24

2.4.4 Futures in AmbientTalk 25

2.5 Component Models and Frameworks 27

2.5.1 Common Object Model (COM) and DCOM 28

2.5.2 Enterprise Java Beans 28

2.5.3 CORBA Component Model (CCM) and GridCCM 29

2.5.4 Common Component Architecture (CCA) 30

2.5.5 Service Component Architecture (SCA) and FraSCAti 31

2.5.6 SOFtware Appliances Component Model (SOFA) 32

2.5.7 Fractal component model 33

2.5.8 Grid Component Model (GCM) 34

2.6 Summary of Related Works and Positioning 36

Concurrency and distribution are widely studied topics in the research communities. A large body of work exists dealing with various aspects of concurrency and distribution. In this chapter, we try to situate our work and present an overview of our research domain. We study efficient transmission of results in distributed systems with futures. As there are no directly competing works on future update strategies[1], we instead focus on works that fall in same research area. We start by evaluating other works that make use of futures or similar constructs as means of providing concurrency and parallelism. We study the various ways in which the futures are implemented/reasoned on, and point out how they relates to our work on future updates. In the later half of this chapter, we present some of the more common component models and the works formalising them (if any), aimed at large scale distributed systems and computational grids. We evaluate these systems with the aim of validating our chosen component model. We formalise a component model for distributed components, with components as units of concurrency, and asynchronous communications with futures. We give a formal semantics for our components that incorporates one future update protocol. The theoretical part of our work focuses on formalisation of such a component model, along with formal semantics for future-update mechanisms.

To the best of our knowledge, our work is the only one which focuses on efficient transmission of future values and attempts to formalise the value-transmission mechanisms for first class futures with the goal of providing mechanised proofs on the properties of update strategies. This serves to prove the correctness of the corresponding implementation of future update strategies in ProActive library [15]. We finish by positioning our work in relation to already existing works and point out why we think that our work can directly benefit other languages and frameworks that use futures.

Finally, we use Isabelle/HOL [12] theorem prover for our formalisation and reasoning. This thesis is focused on futures, components, and the interplay between them. A comparative analysis of various theorem provers is neither the intent of this thesis, nor is very useful in our context. We do benefit from a lot of features provided in Isabelle/HOL, however we believe that the choice of a theorem prover is secondary; similar formalisation and proofs could be achieved using some other higher-order theorem prover like Coq [17].

2.1 Background: Why Futures ?

Distribution and concurrency have attracted a lot of research in computer science. Traditionally, concurrency is achieved by creating separate tasks/processes or threads that execute some part of the code in a concurrent manner. While such construct do allow the programmers to specify which sections of the code may be executed in parallel, they are very intrusive to the business logic. Special attention

[1] [16] presents some experimental evaluation with different future update strategies, and a part of our work extends that paper. Further details appear in Chapter 4.

has to be paid on access to shared resources and synchronisation between the various concurrently executing tasks/threads/processes; ensuring that the behaviour of the concurrent/parallelised program stays correct with respect to the specification.

Futures first appeared as language constructs in functional programming languages to provide a simple and easy to use programming model for concurrent and parallel applications. While the actual details and semantics of futures may vary between various languages/frameworks, most programming languages/frameworks support some variation of futures as proposed in Multilisp [3] and ABCL/1 [4]. However, it should be noted that although Multilisp and ABCL/1 were among the first languages to support and popularise the use of futures, the ideas introduced were not new. Similar ideas on parallel execution of code blocks had already appeared in literature. One important work that proposes a similar idea is Hibbard [18]; the eventual value of Hibbard were roughly equivalent to futures in Multilisp. However, a key difference between the eventual values and a future is the type of return value. The eventual values were represented as a separate type, similar for example, to the future type in java concurrency API [19]. On the other hand, a future in Multilisp has the same type as the result it represents.

2.1.1 Some basic questions about futures

In order to establish the context for a review of works relating to futures, we use the the following paragraphs to answer some basic questions about futures as they are used in various programming models.

What is a future? Futures are programming language constructs providing concurrency and data flow synchronisation. Simply put, a future is a placeholder for the result produced by a concurrent/parallel computation which is yet to produce the result value. Futures introduce concurrency in programs by allowing the current executing thread/process to continue its execution without waiting for the result from a concurrent computation.

How are futures created? Mechanisms for creating futures depend on the particular programming language/framework, however they all fall into one of the two categories; *explicit* or *implicit* creation. Some languages/frameworks require futures to be created in an *explicit* manner. This means that the language/framework contains specific language constructs that allow creating futures. Programmer has to modify his sequential program to explicitly mark which sections of the code should benefit from concurrency. Examples of such languages/frameworks include Multilisp [3], ABCL/1 [4] and ABCL/f [20], Alice ML [21] and $\lambda(fut)$ [5], Creol [6], Java [19], etc. The second option is to have futures that are created in an *implicit* manner. The language/framework itself is responsible for deciding when to use futures to introduce concurrency, or at least for the creation of futures. Examples of such works include ASP [7] and ProActive [15], AmbientTalk [22], etc. In the above mentioned languages/frameworks there are no explicit constructs to

create futures. Instead, futures are created implicitly, as a consequence of invoking asynchronous method calls on remote processes.

How are futures manipulated and accessed? The languages where futures are explicit and exist as a separate type allow the programmer to manipulate futures using a variety of constructs. Usually, three such operations are supported. A future creation operation `future`/`spawn` to create new futures, `touch`/`get` operation to fetch the value when it is available, and `peek` operation to check if the future value is available. On the other hand, the languages/frameworks that use implicit futures make no difference between a type τ and the type of a future that will be filled with a value of type τ (`future` τ). Implicit futures are accessed and used the same way as the results they represent.

What happens to a future when the concurrent execution finishes? When the concurrent task computing the value of the future is finished and a result value is available, it should be transmitted back to the thread/task which spawned this task. However, the various languages/frameworks differ in how future values are fetched. In Multilisp for example, the future value is fetched in an implicit manner. The underlying language compiler itself places the necessary `touch` operations in the program body. On the other hand Creol and Java require the `touch`/`get` operations to be explicitly invoked. Yet another approach is taken by ASP and ProActive, where future are not only implicit but transparent as well. The future value is fetched and the future gets replaced by its value transparently.

What happens when a program tries to access a future for which the value is not yet available? A future for which the value is not yet available, usually because the concurrent computation is not yet finished, is said to be *unresolved* or *undetermined*. When an attempt is made to access the value of such a future, the currently executing task/thread is blocked until the value is available. Indeed this is the behaviour of futures in majority of languages and frameworks, for example Multilisp, ABCL/f, Creol, ASP all block on access to an undetermined future. AmbientTalk is an exception to this widely followed semantics, and future access in AmbientTalk [22] is non-blocking.

To summarise, a future is a placeholder for the result of some concurrent execution, and may be created either explicitly or implicitly. Explicit futures exist as separate type in the language and provide operations to manipulate them; the result has to be acquired explicitly when the concurrent execution terminates. Implicit futures are created by the underlying language implementation, without the intervention of the programmer. Such futures share the same type as the result value they represent. When a result is available, an implicit future may be transparently resolved. Finally, access to an unresolved future is a blocking operation; the execution thread is blocked until the result value becomes available (AmbitentTalk is an exception). Having established a brief context for futures as means of providing

concurrency and parallelism, we now review the various existing works that employ futures.

2.2 Distributed and Concurrent Programming

Distributed and concurrent programming has attracted a lot of attention from the research community. A huge amount of research has been done in the domains of distributed and concurrent programming. It is not possible to summarise all the works in these domains, neither is it very useful in the context of our work. Therefore, in this section we only focus on various programming constructs that have been developed and are used in popular programming languages to support distribution. Similarly, we look at how concurrency is supported in modern programming language and what kind of programming abstractions are provided for concurrency. It should be noted here that distributed by default does not mean concurrent or vise versa. It is very much possible to have a distributed system where processes are blocked waiting for the execution to finish on other machines, hence giving rise to a distributed systems that on the whole or in part acts as a sequential system. Similarly, a concurrent system need not be distributed. Hence, the key element of development of distributed concurrent applications is to maximise the amount of work that can be done concurrently/in parallel. The less time that is wasted waiting for remote processing to finish, the higher is the efficiency.

Traditionally, concurrency is achieved via processes, tasks or threads [23]. However, executing multiple sections of code in parallel raises the question of 'how the concurrently executed processes or threads should communicate with each other'?. Based on the answer to this question, two broad concurrency models may be identified. In *shared memory* model, the various concurrently executing processes or threads have access to some shared memory locations. These locations are used for communication. However, use of shared memory leads to *race conditions* among competing processes/threads. A number of constructs at various levels of abstractions have been proposed to avoid race conditions as well as to avoid situations that can potentially lead to deadlocks. These include various types of locks, semaphores, monitors, etc. The programming languages that support shared-memory concurrency model include, C, C++, Java, Multilisp etc.

As opposed to shared memory model, the *message-passing* communication model ensures that all communications take place in the form of messages. Various existing APIs provide support for both synchronous and asynchronous communications. With asynchronous messages, the sender can continue its execution without waiting for the reply. The support for asynchronous message-passing has given rise to *Actor* [2] or *Active Object* [7] model of concurrency. Simply put, these models support concurrency in the form of parallel processes that communicate via asynchronous messages. Examples of such languages include ABCL/1 [4], Creol [24], ASP and ProActive [7], etc.

Although, threads and processes are widely used in modern languages to provide

concurrency, they are cumbersome to use for implementing large scale distributed systems. Programmer have to be extra careful to ensure that access to shared resources does not cause race conditions. While effective, these are low-level constructs that require careful manipulation.

Futures were introduce to simplify creation of concurrent applications; specially in the context of functional languages without side-effects they provide an easy annotation style construct for converting sequential code to concurrent code. The following section (Section 2.3) deals with how futures are handled in various languages/frameworks.

2.3 Distributed Concurrent Programming with Futures

Futures enable an efficient and easy to use programming paradigm for distributed applications. In the following sub-sections we briefly present how futures are implemented and supported in the various functional and non-functional languages and formal calculi. The goal of this section is to position our work on future update mechanisms in the wider context of how futures are currently implemented and to point out the differences and similarities between our work and other existing approaches.

2.3.1 Futures in Multilisp World

Multilisp [3] along side ABCL/1 [4] was among the first to introduce futures as language constructs to support parallel execution. Multilisp is a dialect of the functional programming language Scheme [25], which itself is a dialect of Lisp programming language[26]. Scheme is a functional language providing features such as lexical scoping of variables, first class continuations, delayed execution, etc. Multilisp extends support for parallel execution and shared memory. Even though Multilisp is a functional programming language, it also contains language constructs that allow *side effects*, i.e., it supports modifications to already computed values. This differs from pure functional programming style, where the output of a function depends only on the value of the arguments passed to it, thus ensuring that two different calls to the same function, with same argument values, always returns the same result.

Multilisp uses the construct `future` to support parallelism. Given an expression `X`, `(future X)`, immediately returns a future which acts as a placeholder for the eventual value of expression `X`. At the same time, a new task is created to start evaluating the expression `X` concurrently. Once the expression `X` is evaluated, the resulting value is used instead of the future. The use of `future` allows the program to continue its execution, in parallel to the evaluation of the expression `X`, instead of waiting for the result to be evaluated. This can lead to significant improvement in the amount of parallelism between code blocks.

Initially, when a future is created, its said to be *undetermined* or *unresolved*; the future has not yet received the computed result value. A future is resolved/determined, only when it receives the computed value. Any operation that tries to

access/use the value of an undetermined future results in the calling task being blocked until the future becomes determined/resolved. Such operations are referred to as *touching* primitives [27]. Once the value is computed for the future, its status is changed to determined and all tasks waiting for this future are notified. In Multilisp, this is achieved by associating with each future, a list of tasks that are interested in the value of the future. When the value is received, all tasks in the list are notified. This allows for data-flow synchronisation. On the other hand, operations such as argument or parameter passing does not need to know the computed value of the future. Therefore in Multilisp these operations can be performed on undetermined futures.

Future in Multilisp are created explicitly. In addition, Multilisp provides constructs like `touch` to fetch the value of a future. Futures are first class entities, i.e., they can be passed as arguments and parameter values. In Multilisp, all tasks share the same address space, which results in a shared memory model. While the future creation is explicit, the computed value is associated with its relevant future in an implicit manner, i.e., futures are resolved implicitly.

Multilisp also supports the concept of *lazy evaluation*, through the use of `delay` construct. The delay construct works the same as `future` construct, except that with delay, the execution is only started when some other computation requires the value, indicated through an implicit `touch`.

Futures and continuations In [27], the authors discuss a key problem which may arise from using the Multilisp future construct with *continuation*s. A continuation refers to the remaining steps to be performed in the execution of a program. When a programmer makes the scheme call `call-with-current-continuation (...)`, the continuation captures the remaining steps to be performed in the program. A continuation contains information such as current program stack and current point in the execution. This continuation is provided to the programmer for manipulation through *reification* [28], and may be used to resume the program from that particular instance. The process of applying a reified continuation is referred to as *throwing* that continuation.

Continuations are a powerful language construct and allow for greater flexibility, however they pose some interesting problems. Continuations allow the execution to return to a previous point multiple times. This would normally not be a problem in languages which do not have side effects. In the presence of side effects, the execution following the throw (resume from continuation) may differ from the original execution. Also, as the control may return to a point in execution multiple times, this means that a future may end up being computed more than once. This violates the *computed only once* principle of Multilisp. The correct semantics in this case is to indicate an error.

The paper [27] identifies two main problems as a result of using futures with continuations. The first is the possibility of having multiple result values assigned to a single future construct as a result of resuming from a continuation. This is

resolved by improving the implementation of the default future construct, ensuring that all returns after the first one are handled in a different manner, ensuring correct behaviour. The second problem deals with the possibility of having multiple values returned by a single program. This is resolved by introducing the concept of *legitimacy* of each thread. A thread is legitimate if the code it executes is the one that would have been executed in a sequential implementation without futures. Only the results returned by the legitimate thread/task are considered while all results from illegitimate threads may be safely ignored. Authors claim that with the two proposed improvements, it is guaranteed that the program which uses both futures and continuations, produces the same results on a parallel machine as would be produced on a sequential machine. However, no formal reasoning is provided.

Formal semantics Formal semantics for futures in Multilisp appear in [29, 30] in the context of an idealised functional language. Four operational semantics are presented with the goal of analysing the programs using futures and optimising the program by eliminating unnecessary touch operations. The first semantic defines future to be a semantically transparent annotation; semantics are defined using a sequential evaluator function – eval_s from program to results. The second semantic validates that a future expression interpreted as process creation is correct. The authors prove this by defining the semantics of a parallel machine through a parallel evaluator function – eval_p – and showing the equivalence of the two evaluators. This ensures that the parallel machine correctly implements the sequential machine in that they both define the same semantics for the source language. The third semantics explicates the coordination of parallel tasks by introducing placeholder objects and touch operation. The last semantic is used to derive a program analysis algorithm, resulting in a touch optimisation algorithm. Authors present the correctness proofs of their algorithm, and show that their algorithm can produce significant speedups using tests carried out on benchmarks. The experiments are carried by incorporating the proposed algorithm in Gambit Scheme compiler [31]. Similarly, [32] presents the formal operational semantics for futures for a scheme-like language that supports both side-effects and first class continuations.

Comments Most of the modern languages that support futures, offer some variant of futures as presented in Multilisp. However, important difference exist; ranging from how and when futures are created, to what granularity of concurrency they represent? Futures in Multilisp can be very fine grained, created only for evaluation of single expressions. In contrast, our work deals with futures that correspond to a more coarse-grained concurrency model, and are created in response to asynchronous method invocations. As opposed to Multilisp, we view futures as both transparent and implicit. The underlying framework is responsible for transparently fetching the value of the future, and replacing the future reference with the actual value. Finally, all Multilisp tasks live in the same shared memory space and can use shared data structures for communication; while our communication model is based on

a asynchronous request-reply paradigm, with no shared memory. Absence of any shared memory leads naturally to a coarse-grained concurrency model that is centred on method invocations.

2.3.2 Futures in ABCL/1 and ABCL/f

ABCL/1 [4] is an object-based language for parallel computing. The computation model of ABCL/1 is inspired by the actor model [2, 33]. Objects in ABCL/1 are autonomous entities, having their own processing power (execution thread) and local persistence memory representing object state. The objects communicate with each other via messages-exchanges. An object in ABCL/1 can be in one of the following modes: *dormant*, *active* or *waiting*. Upon creation, objects are initially dormant. They become active on receiving messages, as defined by the description in the deployment *script*. Each object has an associated script, which defines the messages it can accept, and what actions to take upon receiving them. Once an active object has completed the actions it needs to execute in response to the accepted message, it becomes dormant again–until the arrival of next message. While active, an object may need to wait for the arrival of a specific message for synchronisation. In this case, the active object may go into the waiting mode, until it receives the required message. All the processing by an object takes place while in active mode. An active object may transmit a message to any other known active object, regardless of the mode of the target object.

To facilitate the communication between objects, each object has two message queues associated with it. One is the *ordinary mode message queue*, while the second is for the *express mode message queue*. Express mode message queue is used for messages that require priority handling. ABCL/1 also supports a future type message. This roughly corresponds to future construct of [3]. It should however be noted that the future construct of Multilisp [3] is more fine-grained than the implementation in ABCL/1 (message level). When a future type message is sent, a private future object is created at the sender side which corresponds to the not-yet received result. The sender can then continue its execution until it actually requires the future result. Language constructs are provided to check whether result of a future is available or not. If no result is available, the sender has to go into waiting mode, until the message with the result arrives.

ABCL/f [20] is a further development from ABCL/1. However, unlike ABCL/1, ABCL/f allows for usual function/procedure invocations and method calls as well. Additionally, ABCL/f is a typed language in contrast to the untyped ABCL/1. The basic syntax of ABCL/f is derived from Lisp. Concurrent objects in ABCL/f communicate via asynchronous method invocations. To accommodate this style of communication, ABCL/f provides a object-based variant of the future construct, as proposed by Multilisp [3]. An object can send a message or make a call to a target at any time, regardless of mode/state of the target. On the receiver side, a concurrent object uses a queue to store the calls/messages and treats them one by one. ABCL/f supports three types of invocations, *Past*, *Now* and *Future*. Past

type invocation is purely asynchronous without any result or callback involved, and the invoker immediately continues its execution. For an invocation of type, Now, the invoker send the message (makes the call) and waits for a result to arrive before continuing its execution. Future type invocation, creates a placeholder object for the result on which the invoker waits for the result when needed. After the placeholder object has been created, invoker can continue its execution. When the invoker requires the result, it can perform a `touch` operation on the placeholder. Future object in ABCL/f are treated as a first class object. Touch is a blocking operation that fetches the results for the future object being touched. If the result is not available, the process calling touch operation is blocked. When the value arrives, the invoker is unblocked and can receive the value. Optionally, the `peek` operation may be used to check if the future value has already arrived or not. Each target that receives a future type invocation, implicitly receives the future object. It uses this object to send back the computed value. In addition to this implicit return, ABCL/f also allows for explicitly returning the future value through a explicit `(reply value {:to future })` notation. In case of multiple replies, all replies after the first one are simply ignored. Consequently, the futures in ABCL/f are not strictly tied to a concurrent execution and any process with access to future can produce the value of a given future, using the explicit reply construct.

Comments ABCL/f is a descendant of the ABCL/1 language and is based on the notion of concurrent object that communicate via asynchronous method/function calls. While the programming and communication model of ABCL/f has some similarities with the model used in our work, a number of important differences exist. Futures in ABCL/f are created explicitly, using the `future` keyword. Access to the value of future is also explicit. This differs significantly from our work, which is based on notion of futures which are both transparent and implicit. Similar to our approach, futures in ABCL/f are created as a consequence of asynchronous method calls; this represents a more coarse-grained granularity level than Multilisp. However, future creation in both ABCL/1 and ABCL/f is explicit, whereas we rely on implicit future creation. Access to unresolved futures is a blocking operation in both our work and in ABCL/f, however the implementation differs significantly. ABCL/f futures are resolved through explicit touch operations. A similar future update semantics is supported by our work, however, we are not restricted to this model. We also support other *eager* approaches which allow future values to be fetched as soon as the concurrent asynchronous execution is completed. Therefore, we provide much greater flexibility in the mechanisms used for getting the result values. Finally, ABCL/f supports explicit replies to futures, any process can produce the value of a known future. This is sharply in contrast to our work where each future is associated with strictly one unique process that evaluates its value. Decoupling a future from its value-producer may lead to a situation where multiple values are produced by different processes for the same future. Such a situation is not possible in our case, where only one unique asynchronous method call may produce the result

for a given future.

2.3.3 Futures in Alice ML and $\lambda(fut)$

Alice ML [21, 34] is a typed functional programming language based on Standard ML [35] and Oz [36]. Alice ML extends the standard ML language to support concurrent and distributed programming in a modular manner. The language provides a rich set of features including concurrency, modularity, support for components and high-level support for distribution. The concurrency model of Alice ML is based on futures [3] and Promises [37]. Alice relies on futures to provide a light-weight concurrency model. Futures are created explicitly, and can be of either *concurrent* or *lazy* variety. A concurrent future is created using the language construct `spawn`, resulting in a future whose value is calculated in a concurrent thread. Presence of lazy futures provide support lazy evaluation. A future created using `lazy` allows the programmer to defer the concurrent evaluation until it is actually needed by some process, i.e., computation begins only when the value of the future is requested. An Alice program may utilise both eager and lazy evaluations inside the same program. Futures in Alice are first class entities and can be passed around as values. Similar to other frameworks/languages like Multilisp, ABCL/f, etc., access to a future, which is yet to be computed results in the calling thread/process being blocked, until the future value has been computed. While the future creation is explicit through the use of `spawn` or `lazy`, they are eliminated transparently once the results become available. Once the value of a future is available, it is sent to where it is required. In addition to this implicit synchronisation on future access, the synchronisation can also be done explicitly by using the `await` construct.

Alice ML also supports *Promise* to provide explicit and more fine-grained control over creation and resolution of futures. A Promise is an explicit handle to the future, and provides a decoupling between future creation and resolution. Each promise has an associated future, which can be accessed by programmer through the promise. A Promised future is not replaced automatically but has to be resolved explicitly through a `fulfill` function. However, fulfilling a promise does not mean that its associated future is also determined. A promise may be full-filled by any thread with any future, unlike simple futures which may only be computed by their associated asynchronous thread. Finally, a promise may be full-filled only once.

Formal semantics A formal semantics for Alice ML is presented in [5] in the form of a lambda calculus with support for futures $\lambda(fut)$. $\lambda(fut)$ provides *concurrent futures*, and *handled futures*. Concurrent futures correspond to the futures in Alice ML, except they permit recursive use of future in the evaluating expression. *handled futures*, are similar to the promise construct of Alice ML. Handled futures provide a once-only write permission for the future value with a given handle. The future values may be computed in eager or lazy manner. However, the computed values are fetched on need. The authors present a static type system for $\lambda(fut)$ and identify a confluent fragment of $\lambda(fut)$ which they prove to be uniformly confluent.

A liner type system (type system for single-reference objects; useful for immutable datatypes like futures) for $\lambda(fut)$ is presented. Authors show that the system is rich enough to type definitions of various concurrency constructs used in Alice ML, with the aim of showing that they cannot be corrupted in a well typed context.

Comments Alice ML and its formalisation $\lambda(fut)$ uses a concurrency model where all synchronisations are data-flow synchronisations, and are based on futures. This is somewhat similar to our currency model, except our concurrency model is more coarse-grained. As opposed to Alice, we only allow execution of asynchronous method calls in parallel. Also, we view futures as transparent and implicit. Unlike our model, Alice requires the creation of futures to be explicit. Access to future is similar to our model, and future values are fetched implicitly. Access to an unresolved future is similarly blocking. Alice ML also provides the promise construct, providing more fine-grained control over future creation and resolution. Each process has an associated future. However, unlike simple futures, a promise may be full-filled by any thread, with any future. This creates the possibility of leaving the future associated with that promise undetermined. In our work, we use futures that are associated with only one unique asynchronous method call. Our futures can only be determined once, i-e., there is only one value corresponding to a future and only one unique concurrent execution may produce that value.

$\lambda(fut)$ provides a type system for type definitions of concurrency constructs. The future values are computed in either eager or lazy manner, but the results are fetched in a lazy manner, i.e, futures are resolved only when their value is needed. In contrast, our work supports different future update strategies, allowing the results to be fetched in either lazy or eager manner. We focus on providing semantics of future update mechanisms, and on having first proofs on correctness of these mechanisms without providing a type-system. $\lambda(fut)$ on the other hand does not deal with the exact semantics of future update mechanisms.

2.4 Distributed Concurrent Programming with Futures and Objects

Object-oriented programming is now a well established part of distributed and concurrent programming landscape. In this section we present the more popular object-oriented approaches which support concurrency through futures.

2.4.1 Future in Java-verse

Java [38] is a popular object oriented programming language, developed at Sun Micro-systems (now Oracle), that is widely used in distributed systems. Support for futures was introduced in Java as part of `java.util.concurrent` package, with the goal of facilitating light-weight concurrent programming in java. Futures in java are explicit; i.e., the programmer has to use explicit programming construct to create

and manipulate futures, similar to Multilisp, ABCL/f, and Promise in Alice ML. The key class providing support for futures in Java is the `FutureTask` class, containing a number of useful methods for manipulating futures. For example, it provides methods like `isDone( )` and `get( )`, corresponding to `peek` and `touch` operations as specified in [20, 29]. Access to future value is also explicit. Programmers can attempt to acquire the future value using a `get( )` operation, which blocks until the value becomes available. Another interesting characteristic of Java's implementation of future is the support for *cancelling* the concurrent computation. The `cancel( )` method of the `FutureTask` class allows the programmer to attempt cancelling the concurrent execution. Only computations that have not been completed may be cancelled. The method `isCancelled( )` provides a feedback mechanisms to the programmer to check if the task was cancelled successfully before completion or not.

Other works Futures were first devised for improving concurrency in functional programming languages. In the context of a functional programming language, it is expected that a program with futures will provide the same results as a program that is executed sequentially. This is made possible by the lack of side-effects and mutation. Java is an imperative language where frequent access to shared memory containers is made and results are obtained and updated via mutation of container objects, like arrays and heaps. The current java implementation does not prevent problems arising from access to shared memory containers. The programmers have to ensure that concurrent access due to futures, does not violate correctness properties.

In [39], the authors present a *Safe Future API*, which ensures that injection of futures in an existing serial program does not violate any existing data dependencies. This is achieve through a compiler and run-time infrastructure, that relies on Object versioning and task revocation techniques to identify safety violations and remedy program execution when such violations are detected. Experimental results are provided to evaluate the performance overhead of safe futures.

In [40], introduces a framework for simplifying concurrent programming in java through the use of transparent proxy objects. Static analysis is used to allow the programmer to use future without making invasive changes to the program for satisfying type restrictions. The flow of futures in the program is tracked, and *claims* are inserted in the byte code where the actual value is required rather than the future variable itself. A claim is essentially a *touch/get* operation, which may cause the current thread to be blocked until the value is available. The analysis is based on qualifier inference, and is formalised to prove its soundness. Experimental evaluations are carried out to test the framework.

Directive-based lazy future [41] aims to further simplify the usage of futures in java. The framework makes use of Java's annotation mechanism to label the local variables that receives results from function calls that may be executed in parallel. The annotation mechanism makes it easier for the programmer to use futures for exploiting parallelism in previously sequential programs at the cost of little extra

overhead, and is much closer to how futures are used in functional programming languages. However, unlike side-effect free functional languages, programmer still needs to ensure that any race conditions between concurrently executing threads is handled correctly. An extension to the framework for handling exceptions and failures is presented in [42].

Comments Java is a widely used programming language; a number of works exist on futures in java. However, all of the works presented above deal with explicit futures. Programmers explicitly specify which tasks/functions should be executed concurrently. Similarly, futures are accessed and resolved explicitly; this differs from futures in our work which are implicit and transparent. Futures in our work are created in response to asynchronous method invocations, and have the same type as the type of result they represent. Our approach treats future of a result of type X the same way as a variable of type X. This means no further changes are needed to the methods which might receive futures as arguments or return futures. On the other hand, as future is a explicit type in java, method signatures have to be changed to enable them to receive futures as arguments or return values. Finally, while approaches like Safe future API attempt to resolves problems introduced due to shared memory, this still involves overheads in java. We deal with a message-passing only communication paradigm with no shared memory, thus removing the problems arising from shared memory.

2.4.2 Futures in Creol

Creol [24, 6] is a high level object-oriented programming language, addressing concurrency and distributed systems. The language is based on the notions of concurrent objects typed by behavioural interfaces, that communicate by asynchronous method calls and processor release-points. The asynchronous method calls are supported through futures. Active objects in Creol may be multi-threaded, although only one thread may be active at any given time. The various intra-object threads cooperate explicitly using *processor release points*, which essentially are `await` statements. This allows the currently executing thread to assume that no other thread is accessing the object's attributes, leading to a programming and reasoning model resembling monitors [43]. Futures in Creol are explicit. Specific language constructs are provided for creating futures. Additionally, futures are non-transparent, i.e., futures have a separate type than the value they represent. In order to manipulate futures, explicit operations like `touch` are used. Creol uses the `get` operation to allow all processes interested in the future value, to register as observers. Once the value of a future is available, all observers are notified. The future-value retrieval mechanism of Creol is somewhat similar in concept to ABCL/f and Java futures.

Creol also offers components; the paper [44] presents a framework for component description and testing. A simple specification language over communication labels is used to enable the expression of the behaviour of a component as a set of traces at the interfaces. Creol's component model does not support hierarchical structure of

components. In [45], the authors present a formalisation of the interface behaviour of Creol components. Creol's operational semantics use rewriting logic based system Maude [46] as a logical support tool. The operational semantics are expressed in Maude by reduction rules in a structural operational semantics style, enabling testing of model specifications.

Formal semantics A number of works exist on formalising various aspects of Creol. In [6], a nominal type system (equivalence of types is determined on name of types) is introduced for Creol programs. The authors show that execution in objects typed by behavioural interfaces and communicating by asynchronous method calls is type safe. Additionally, the paper also presents an executable semantics for Creol. A formal semantics for an object-oriented language based on Creol is presented in [47], comprising notions like active objects, asynchronous communications and futures. The paper extends the core language and adds support for first class futures. The authors provides operational semantics for the core Creol language, with first class futures, using an extension of Featherweight Java [48]. The concurrency model allows multiple concurrent threads to exist inside each active object, although the Creol constraint of having only one thread active at a time is maintained. Asynchronous method calls are used to trigger concurrency, each call results in a method being executed by a concurrent thread in the target object. Each object holding a shared future, may completely block for the future, or it may use the `await` construct to release control only for the thread awaiting the value for the future. In Addition to the operational semantics, a formal proof system for proving properties relating to concurrency is also presented. The proof system relies on a two-tier assertion model. A local assertion language is used to describe the local state of the object in monitor invariants (monitoring the release points) for intra-object synchronisations. While a global assertion language describes the invariants for inter-object synchronisation.

Comments Creol is similar to our work in the sense that both comprise similar programming constructs and notions; such as, concurrent active objects, asynchronous method calls and futures. However, there a number differences between our work and Creol. In contrast to our work, futures are created explicitly in Creol. Also, future manipulation and access is also explicit and controlled by programmer. On the other hand, we deal with futures that are both transparent and implicit. Creol has been extended to support first class future, similar to futures in our work. However, the works on Creol do not explicitly specify the different update mechanism for first class futures. We focus on future updates and provide specification and implementation of three main future update strategies. This work can potentially be applied to Creol as well. Finally, our formalisation supports concurrent execution of multiple requests (asynchronous method invocations) as opposed to 'only one thread active at a time' model of Creol. However, our real implementation is in ProActive [7] which only allows one execution thread per object; other threads may

be created to deal with tasks such as result communication, etc.

2.4.3 Futures in ASP, ProActive and ASP$_{\text{fun}}$

The *Asynchronous Sequential Processes* (ASP) [49, 7] is an imperative distributed calculus, derived from $\varsigma-$*calculus* [50], and models an object-oriented programming language based on the notions of active objects that communicate via asynchronous method calls with transparent first class futures. However, each active object itself is sequential and contains only one execution thread. In ASP, an object can either be an *active* object or a *passive* (simple java) object. Each activity wraps a single active object, which may contain other passive objects. Activities may communicate via asynchronous method calls (requests). Any request sent to an activity is in fact sent to its active object. The activity serves each of the received requests one after the other. All received requests are stored in a pending request queue. Communication between activities is solely based on asynchronous method calls and there is no shared memory between active objects. Any passive object passed as arguments of method calls or return values are passed using copy-by-value semantics. In contrast to Creol, activities in ASP are mono-threaded, i.e., there is only one request being served at a given time. ProActive distributed programming library [7, 15] may be considered as one *possible* implementation of ASP-calculus.

One of the main features that distinguishes ASP and ProActive from other works is the support for transparent implicit futures. Futures in ASP and ProActive are completely transparent and there are no explicit instructions to manipulate futures. Futures are created as a result for asynchronous method invocations, i.e, an invocation on an active object. From a programmer point of view, the remote invocations are expressed in the same manner as local invocations, and there is no change required in the code. Familiar '.' notation is used for both local and remote invocations. Additionally, a future for a type X may be used in all expressions and locations where a value of type X may appear. Again from a programmer point of view, there is no distinction between a future and a value. There are no specific operations for fetching the future value. Futures are first class; they may be communicated between active objects. Access to an unresolved future is a blocking operation; the calling execution thread is blocked until the result for that future is available. This is referred to as a *wait-by-necessity*, and is an implicit data-flow synchronisation on the future. Wait-by-necessity occurs when an active object tries to access a future for which there is no result available yet. At some stage the execution of the request – corresponding to that future – terminates and a result is produced. The produced result is sent back to the caller (which received the future) where the result transparently replaces all occurrences of the future object. Unlike futures in java, future in ASP and ProActive are resolved implicitly, without the need of a *touch* or *get* operation. ASP allows the possibility of having partial results and replies; the result value may itself contain other futures.

In [7] a number of possible mechanisms for retrieving the value of a future are

discussed. ProActive implements[2] one such approach while an initial implementation of other approaches appear in [16]. Our work may be considered an extension of [7, 16] through a language independent approach that makes it applicable to various existing frameworks with futures. We improve upon the initial implementation and provide some experimental results on performance of future update strategies. The details on futures in ASP and ProActive appear in Chapter 4.

Formal semantics ProActive may be considered as a possible implementation of the formal model presented in ASP-calculus. The model defines a small step semantic for ASP using parallel reduction rules, in addition to well formedness rules. To manage the complexity of reasoning about distributed and concurrent systems, the presented language is restricted to a subset of features to insure that the reductions are confluent and deterministic.

ASP_{fun} [51, 52] is a complimentary calculus to ASP, and presents a distributed calculus which uses some of the same notions as ASP, for example, activities which are units of concurrency and distribution, communication via asynchronous method invocations, and first class futures. However, unlike ASP, ASP_{fun} is typed and presents a functional version of ASP, i.e, in ASP_{fun} there are no side-effects. Method calls operate on a copy of the objects passed to them. Finally, ASP_{fun} model and its properties are formalised and proved mechanically using Isabelle/HOL theorem prover [12], whereas properties on ASP were mostly proved by hand.

Comments ASP and ProActive provide an easy to use programming paradigm for concurrent and distributed programming. It can be stated that our work is an extension of ASP and ProActive and aims to extend the future update mechanisms proposed there. For example, ASP discusses the possible update strategies but does not provide a formalisation of how they work, nor does it study the properties of future update mechanisms. On the other hand, ProActive only provides one future update strategy. ASP_{fun} is closely related to our work on formalisation of future update mechanism. It formalises functional language featuring active objects, asynchronous communication, first class futures and a type system. However, although ASP_{fun} supports first class futures, it does not incorporate formalisation of various future update mechanisms, which is one of the goals of our work.

2.4.4 Futures in AmbientTalk

In [8, 22, 53], authors introduce a new language AmbientTalk aimed at loosely coupled small devices communicating over an adhoc network. The language aims to address four key phenomena that are inherent to mobile adhoc networks. These are connection volatility: the connection is not stable; Ambient resource: remote resource may become dynamically unavailable, possibly due to movement of the user; autonomy: every device acts as an autonomous unit, there may not be a

[2] See Chapter 4 for details on future update in ProActive, and how our work provides additional mechanisms for future updates. This work in published in [10, 9].

server or *client*; and concurrency: in order to maximise concurrency, waiting for a result should be avoided. To overcome these limitations, AmbientTalk adopts a concurrent programming model based on [4]; active objects that communicate by asynchronous message passing. Upon reception, the messages are scheduled in the receiver's message queue and are treated one by one. Similar to ASP and ProActive, there is only a single thread of execution per active object. Also, the model differentiates between active and passive (normal) objects. Each active object may contain other passive objects. Passive objects can only be accessed by the wrapping active object to avoid shared objects between active objects, and can be passed between active objects using a deep-copy semantics. The language allows for both synchronous –using '.' notation– as well as asynchronous invocation –using 'o←m( )'– on a passive object. Asynchronous invocation on a passive object is enqueued in the encapsulating active object. For active objects, all invocations are asynchronous.

The communication model adopted by AmbientTalk is based on E [54], and relies on asynchronous communications. All communications between active objects are asynchronous. An active object may continue to send messages to another active object even if the target is unavailable at the time. The messages are queued and are delivered when the object is again reachable. In contrast to a single message-queue of ABCL/1 and ASP, active objects in AmbientTalk compose up to eight different message queues, referred to as *mailboxes*. These mailboxes are used to track various types of messages; for example, outgoing messages(separate mailboxes for messages that are received /not received), incoming messages (mailboxes for received and processed /not yet processed messages), etc. Similarly, there are separate mailboxes for allowing collaboration with other active objects (broadcast messages, etc).

An interesting feature of AmbientTalk which distinguishes it from other works on futures like Multilisp, ASP, Creol, etc., is the way futures are accessed. The future access is completely asynchronous, and allows to transparently forward any messages sent to a future, to its resolved value. The messages sent to a future are forwarded once the future is resolved using a `when (afuture,closure)` construct. Closures are blocks of code that are applied to a future once it is resolved.

Comments AmbientTalk uses a concurrency and communication model that is somewhat similar to ASP, ProActive and Creol. However, unlike these frameworks/languages AmbientTalk is specifically aimed at mobile devices. The main differentiating feature of AmbientTalk from our work is the way futures are accessed. Unlike our work, future access in AmbientTalk is non-blocking. This lack of synchronisation in theory, ensures that there are no deadlocks. However, there are no formal semantics available for AmbientTalk and thus this property cannot be formally proved. AmbientTalk also differentiates between asynchronous and synchronous invocation; each type of communication uses different notation (only asynchronous messages can be sent to active objects). In contrast, ASP and ProActive make no distinction (syntactically) between the two types of calls. All method invocations use the

same '.' notation. ProActive ensures causal ordering of request messages, which is not guaranteed in AmbientTalk. Finally, the asynchronous communications in ASP and ProActive are implemented with rendezvous style; the sender is blocked until the request/message is received at the target. No such requirements exist for asynchronous communications in AmbientTalk.

2.5 Component Models and Frameworks

Component modelling is a vast domain of research and a number of component models have been proposed over the years. A significant amount of semi-formal and formal works exist in literature, supporting the various component models. Due to the vastness of the domain, an exhaustive treatment is neither possible nor the intent of this chapter. We only cover the more well-known component models in the research and industrial domains, with the intent of introducing them in sufficient detail to be able to validate our choice of (a subset of) Grid Component Model (GCM) for our formalisation.

Why components and component models? Components provide an easy to use programming paradigm allowing for better reusability of application code. Component models focus on program structure and improve reusability of programs. In component models, application dependencies are clearly identified by defining interfaces (or ports) and connecting them together. Component have strictly defined structure (COM, DCOM are an exception). This strict adherence to structure make components ideally suited for structural reasoning. This structure can also be used at runtime (not supported in all component models) to discover services or modify component structure, which allows for dynamic adaptation; these dynamic aspects are even more important in a distributed setting. Since a complete system restart is often too costly, a reconfiguration at runtime is mandatory. Dynamic replacement of a component is a sensitive operation. Reconfiguration procedures often entail state transfer, and require conditions on the communication status. A suitable component model needs a detailed representation of component organisation together with precise communication flows to enable reasoning about reconfiguration.

In the following, we give an overview of some of the more common component models used for developing software applications for distributed computing. We start with models like COM, DCOM and Enterprise Java Beans which have been developed in industry and enjoy widespread usage. We discuss why these models are not particularly suited for large scale distributed systems, and in particular for Grid computing. Moving from those purely industrial models, we review CORBA Component Model (CCM), Common Component Architecture (CCA), and Service Component Architecture (SCA). Oriented towards industry, significant amount of research work exists on these models. We contrast those models with our formalised component model. SOFA and SOFA 2 component models provide a good example of some of the academic component models. We discuss the similarities and differences

from our approach. Finally, we present Fractal Component Model and its grid extension, the Grid Component Model(GCM). Fractal and GCM have inspired lot of interest in research community, in particular the grid computing community. The two models share a lot of similarities; we only cover the points which are significant to our work. We discuss why in our opinion, GCM is a good candidate for our formalisation.

2.5.1 Common Object Model (COM) and DCOM

The Common Object Model (COM) [55] was developed by Microsoft in the 90's and still enjoys wide usage today. COM is a binary standard and as a result is not tied to any particular language. It specifies how components interact at a binary level by specifying a number of key services. COM does not attempt to define what a component or an Object is. Rather a component is defined in terms of three key concepts. A component is uniquely identified and can implement multiple interfaces. Each implemented interface is uniquely identified and is immutable, i.e, the component interface may not change. Components provide a discovery mechanism through which the various interfaces (services) may be discovered. As COM is a binary standard, there is no specification on how the component should be structured internally. The aim of a well-defined COM component is to allow the reuse of the component without any knowledge of the internal structure, which emphasises the importance of clear and precise definition of component interfaces. The Distributed Component Object Model (DCOM) [56], is the distributed extension of COM. DCOM adds support for Remote Procedure calls to COM and solves the additional issues that arise due to distribution. These include marshaling/unmarshaling of method arguments and return values, distributed garbage collection, and distributed discovery service.

Comments Although COM and DCOM both enjoy wide usage on Microsoft windows platform, neither are particularly suited for large scale distributed systems. Both are aimed primarily at desktop application development and do not address the needs of high performance distributed computing; primarily because they lack support for efficient parallel communication between components. COM does not support popular object oriented constructs such as inheritance and polymorphism. Although, support for inheritance may be added through aggregation, it is inefficient and cumbersome to business logic.

2.5.2 Enterprise Java Beans

The Enterprise Java Beans (EJB) [57] is a server-side component architecture for java platform, developed by (former) Sun Micro-systems[3]. The EJB specification aims to provide a standard way to implement the server-side code; the programmer focuses on implementing the business logic or the functional code, while non-

[3]Sun Micro-systems has since then been acquired by Oracle.

functional services like persistence, fault-tolerance, transaction support, life cycle management, security, etc., are provided by the application container. The EJB model makes a distinction between two types of server-side beans (components): The *session beans* which could be *stateless*, *stateful*, or *singleton*; and the *message driven beans*, which provide support for event-based programming model. All the server side business code is implemented using these two types of beans. A third type of component *entity beans* was used to represent component state stored in persistence storage; entity beans have now been removed from the EJB specification (3.0). The Enterprise Java Beans target large scale transaction-intensive enterprise scenarios, with a special emphasis on scalability.

Comments Although, the EJB architecture is designed with distribution in mind, it is more focused on enterprise applications, with components deployed on server-side application containers. The EJB model only allows components/beans to be implemented in java, and no other language is supported. In addition, the model lacks support for interoperability with the various other existing component models for distributed systems. The model is also not suitable for large scale distributed systems, in particular when the available resources belong to different administrative domains. EJBs also does not support important design features like component composition (component hierarchies) or dynamic features such as runtime configuration-reconfiguration.

2.5.3 CORBA Component Model (CCM) and GridCCM

The CORBA Component Model or CCM [58] is a standard defined by Object Management Group (OMG) [59]. The CCM specification provides business components that are distributed, heterogeneous, and are programming language and platform (operating system) independent. The component specification clearly separates functional and non-functional aspects. CCM supports a complete cycle of component definition, production, deployment, and execution through four models. These include, abstract, programming, deployment and execution models. The CCM abstract model allows the developers to define component interfaces and properties of components. The component communication takes place through various types of *ports*. Two synchronous ports, *facet* and *receptacle* define what services the component provides (facet) and what services it requires (receptacle). CCM components may also generate events via *event source* port, while they may receive event notifications on *event sink* ports. The CCM programming model part defines the Component Implementation Definition Language (CIDL) and is used for describing the component structure, system requirements, implementation classes, component state, etc. The CCM deployment model focuses on how components are deployed; components are deployed in the form of software archives containing the component code and a Open Software Description Language (OSDL) descriptor. Lastly, the CCM execution model focuses on the containers as run-time environments for CCM components. Containers provide a variety of non-functional services like persistence,

transaction, security, etc.

The CCM model is quite detailed and provides a solution to dealing with various problems encountered in large scale distributed systems, particularly grids. It supports language heterogeneity, dynamic connection and disconnection of components, component versioning, etc. The model does not support parallel entities encapsulated into components. Similarly, it lacks support for runtime features such as component reconfiguration.

GridCCM In [60, 61], authors introduce an extension of the CORBA Component Model, aimed at overcoming some of the perceived shortcomings of CCM. The extended model, named GridCCM, introduces the concept of a *parallel component*, which encapsulates sequential components. A GridCCM component can offer parallel execution of all or some of the services it offers. The model supports SPMD (Single Program Multiple Data) style parallel codes. In SPMD code, each process executes the same program but on different data set. The presented framework aims to achieve this with minimal modification to the parallel codes. The definition of the parallel component is provided in an XML file, which describes the parallel methods of the component and how data arguments are distributed among these methods. To support data distribution, a new layer is introduced into CCM model to transparently manage data parallelism. This layer is automatically generated by the compiler using the XML descriptor file defining the parallelism and the IDL (Interface Description Language) descriptor, defining the component. GridCCM makes use of PadicoTM [62] communication layer to support multi-point communications.

Comments The CORBA Component Model CCM and particularly its extension for parallel components GridCCM is an interesting model with respect to large scale distributed systems and computational grids. It supports the heterogeneous nature of grid resources and provides specifications of models concerning component definition, deployment and execution. However, the models does not support reflective features, which are required for dynamic component reconfiguration.

2.5.4 Common Component Architecture (CCA)

The Common Component Architecture (CCA) [63, 64] is a component model driven by CCA forum [65], aimed specifically at scientific components, targeting parallel and distributed infrastructure. The key focus areas for the model are high-performance and federation of resources. CCA is a set of specifications that describe various aspects of the model; for example a scientific interface definition language (SIDL), and the ports that define the communication model. The idea of SIDL is drawn from the CCM model. CCA model does not impose a runtime environment for execution of components, making CCA model portable across a wide range of platforms. The specifications consist of a general model which specifies the component, the framework which executes these components, and ports that form the access

points for components. Components are in fact software modules, with strictly defined client and server interfaces. CCA allows the runtime assembly of components using scripts that interact with the CCA framework, for example Gscript presented in [66]. Ports can be added, removed or connected to other components at runtime. A number of CCA implementations, tailored to specific needs are available. Once such implementation CCA Toolkit (CCAT) is presented in [67]. The toolkit is designed to operate over a number of grid middleware systems, including Globus [68]. This is achieved by exposing various grid services like authentication, discovery, creation, etc., themselves as CCA components. An XML extension to IDL is presented to show how the concept of an Interface Description Language (IDL) may be extended for describing software components. It permits component description and deployment using an XML descriptor.

Comments The intent behind models like CCA is the efficient building, connecting and running of components. However, CCA does not support component hierarchies ,nor does it provide support for reflective features. The CCA components cannot be statically typed. Finally, no formal operational semantics are available for CCA components, thus making it impossible to reason on the component model or its implementations.

2.5.5 Service Component Architecture (SCA) and FraSCAti

The Service Component Architecture (SCA) [69] is adapted to Service Oriented Architectures (SOA) [70] . It enables modelling service composition and creation of service components. SOA provides a way for exposing coarse-grained and loosely coupled services which can be remotely accessed. However, this approach does not address how these services should be implemented. SCA model defines a component model for SOA applications. The main entities in SCA are components that communicate via interfaces. An interface may be provide-interface (server) or require-interface (client).

SCA model supports component composition; a component may be *service* component (primitive) – providing some functionality, or a *composite* – assembled from other components. The model supports implementation of primitive components using languages such as Java, C++, Bpel and spring. The specification also allows mixing of components implemented in different languages in the same application. Composite components in SCA are configured and assembled using a XML-based assembly language, referred as *Architecture description language* (ADL). In addition to services, references and components, the other main SCA abstractions are wires and properties. Wires are essentially connections between services (service interfaces) and references (client interfaces). In contrast to Fractal [71], SCA is a concrete specification with a precisely defined syntax of its ADL, semantics etc. The SCA specifications are centred around the task of describing the assembly and configuration of components which compose the application. This assembly is provided as input to the deployment service which then creates the components and

instantiates the application.

FraSCAti SCA specification does not address the issue of runtime management of the application, for example, runtime monitoring of applications and reconfiguration. These issues are addressed in the FraSCAti Platform [72, 73]. The FraSCAti platform supports an extended SCA component model, where components can be equipped with reflective capabilities for their introspection, monitoring, control and dynamic configuration. This is achieved by using interception techniques for extending the SCA components with non-functional services, which are themselves implemented as SCA components.

A number of approaches exist on applying semi-formal and formal methods to service oriented architectures and in particular to SCA. One such example is the European Union SENSORIA project [74], which proposes Architectural Design Rewriting to formalise development and reconfiguration of software architectures using term rewriting [75].

Comments The SCA model attempts to merge distributed component based programming paradigm with Service oriented paradigm. SCA model specifies a programming model which is independent of any particular programming language or communication protocol used. Additionally, the model supports hierarchical composition of components. However, there is no support for reflective features such as introspection, monitoring or dynamic reconfiguration in SCA model. These shortcomings are somewhat addressed in the FraSCAti model where SCA is extended to support these missing features. The FraSCAti model is built upon Fractal component model making it close to GCM. Due to the similarities between the two component models, our formalisation can be considered as a good approximation of formalisation of FraSCAti implementation as well.

2.5.6 SOFtware Appliances Component Model (SOFA)

The SOFtware Appliances component model [76] is aimed specifically at addressing the problem of dynamic component update at runtime, by providing a component model with strong support for versioning. This is achieved by separating the component interface from the component architecture. SOFA architecture supports component composition and component hierarchies.

SOFA 2 [77, 78] is a further enhancement of SOFA, and extends the base model to support dynamic reconfiguration of hierarchical components, better modelling of the control part of the component, and support for different communication styles. An application may be viewed as a hierarchy of components. Therefore, a SOFA 2 component can either be a primitive component or it can be a composed component, which essentially means that the component is composed from a number of subcomponents. All business logic is contained in the primitive components. A SOFA 2 component is described by its *frame* and its *architecture*. The frame defines the component interfaces, indicating which services are provided by the component,

and which services it requires. The component frame can then be implemented by one or more architectures. The architecture of a composed component describes the structure of the component by instantiating direct subcomponents and specifying the subcomponents interconnections via interface *ties*. The architecture defines the first level of nesting in a component hierarchy. A *tie* which connects two interfaces is realised via a connector. A connector [79] is the mediator of an interaction between components, and establishes the rules that govern component interaction. Connectors have protocol specifications defining their properties. For convenience, the simple connectors, expressing procedure (method) calls are implicit so that they do not have to be specified in an architecture specification. The whole application's hierarchy of components (all levels of nesting) is described by an application deployment descriptor. Components are bound through connectors, which are first-class entities in SOFA 2 [77] and allow the connections to be modelled in an extended way, for example by providing support for different communication styles. A particularity of SOFA 2 is that behaviour protocols [80] are central to the model, to the point that typing relationships are defined by behaviour protocols.

Comments SOFA2 is a academic component model as opposed to models like CCM and CCA which enjoy wide industrial support. The model is close to the Fractal component model as it is aimed at the similar kinds of applications, providing support for hierarchical components, separation between control and business part of the components and dynamic runtime reconfiguration.

2.5.7 Fractal component model

The Fractal component model [81, 71], is a modular and extensible component model proposed by INRIA and France Telecom, that can be used with various programming languages to design, implement, deploy, and reconfigure various systems and applications. To accommodate these aims, Fractal specification provides support for hierarchical components, shared components for modelling shared resources, introspection for monitoring runtime components, and configuration-reconfiguration support. In contrast to component models like SCA, Fractal only provides an abstract component model. This means that the Fractal specification only defines the general concepts and different implementations can be derived from those specifications. Fractal provides an *Abstract* API that can serve as a road map for a real implementation.

The main abstractions in fractal are interfaces, components and bindings. Interfaces in fractal model may be client or server, which corresponds to requires-interfaces and provides-interfaces in SCA and SOFA2. As fractal is a hierarchical component model, components may either be primitive (leaf level components) or composites (composed of other components). Bindings are the connections between components and correspond to connectors or wires. Fractal model requires that all bindings must be explicitly defined. Additionally, in contrast to SCA, fractal model further divides a component into functional and non-functional parts. The

functional part is the component *content* part; while the non-functional part is the control-part, referred as the component *membrane* in Fractal model. The membrane manages the non-functional aspects and can intercept calls on the functional interfaces. All interfaces however belong to the membrane. The interfaces themselves are classified either as a *functional* interface or as a *non-functional* interface. Functional interfaces expose the underlying services through client or server interfaces, while the non-function interfaces are server interfaces providing aspects like binding, introspection etc. Finally, *external* interfaces refer to a client/server interface exposed by a component to other components. For a composite component, there are also *internal* interfaces, exposed only to its subcomponents. A method call arriving at the external interface of a composite component is passed on to the subcomponent via internal interface of composite component. Another key aspect in which Fractal differs from SCA is the notion of *shared* components that represent shared resources. A shared component in Fractal is a component that appears as subcomponent of various composite components. Such shared compositions are not possible in SCA.

The formal semantics for Fractal component model are given in [82]. The authors present a formalisation of the structural aspects of Fractal specification in Alloy [83], a formal language based on first order logic. The consistency of resulting formalised models can be verified through the automated Alloy Analyser, to ensure safety of component applications. The work is aimed at clarifying the ambiguities in the informal Fractal specification.

In [84], the authors focus on verification of the behaviour of a component-based application. They provide tools to specify the behaviour of a component-based specification and verify that the application behaves correctly. The work aims at proving the properties of a specific application.

Comments The Fractal specification defines only an abstract component model, resulting in a number of different implementation like Julia [85] and Dream [86]. Julia is targeted towards non-distributed applications while Dream is a library built using Julia specifically aimed at building message-oriented middle-ware. Part of our work dealing with formalisation of component model is closely related to the GCM model, which is a grid extension of Fractal component model. The formalisation of alloy presented in [82] is oriented towards consistency checking of formalised fractal specification. The approach uses a model-checking and counter-example generation approach to ensure consistency. In Comparison our work is oriented towards studying the interplay between the components and futures; we consider asynchronous components and focus on the dynamic component behaviour. This is crucial when specifying future management procedures.

2.5.8 Grid Component Model (GCM)

The Grid Component Model [11] was proposed by the European Network of Excellence CoreGrid as an extension to the Fractal component model, aimed specifically at grid applications. It extends the Fractal component model to address

grid-specific issues. These include, deployment, scalability, autonomic behaviour, and asynchronous communications. GCM inherits most of the component structure from Fractal. Thus GCM supports hierarchical component structure. It also benefits from Fractal features like separation between functional and non-functional concerns, separation between an interface and its implementation, and extensibility.

A GCM component can either be a primitive component, encapsulating business logic, or it can be a composite component, comprising one or more subcomponents. Similar to Fractal, a component is divided into two parts–functional (content) part and a non-functional (membrane) part. Access to a GCM component is through its declared interfaces. The components have client-interfaces (requires-interface) for emitting messages/invocations, and server-interfaces (provides-interface) for receiving message/invocations. Bindings connect a client-interface to a server-interface. Similarly, the concept of internal and external interfaces is also inherited from Fractal. An architecture description language ADL may be used to describe the architecture of a GCM component. The membrane of a GCM component also has a component structure, as discussed in [87]. Unlike Fractal, in GCM the controllers inside the membrane of one component may communicate with the membrane of other components through bindings between non-functional interfaces.

GCM supports a variety of communication models, with concrete choices left to the specific implementation. Therefore implementations may choose between any communication model, for example synchronous, asynchronous, event-based, etc.

A number of works have focused on formalising different aspects of the GCM component model. In [88, 89] the authors focus on the verification of the behaviour of a component-based application. They provide tools to specify the behaviour of a component application, and verify that this application behaves correctly. Their model is applied to the GCM component model too but they prove properties of specific applications whereas we formalise the component model itself.

A formalisation of GCM appears in [1, 13][4]. The authors formalise a subset of complete GCM specification. The model comprises of hierarchical components that communicate via asynchronous method calls with futures. Components are the basic unit of concurrency, and communicate only through asynchronous method calls; there is no shared memory. Asynchronous communication is achieved using futures; method invocations are enqueued at the target component, while the invoker receives a future. Based on the above notions, authors present a a functional reduction semantics for their component model in Isabelle/HOL which is used for deriving mechanised proofs. The reduction semantics are used to prove the correctness of ProActive-GCM [90], which may be considered as an implementation of this model. More details on the presented formalism appear in Chapter 5 and Chapter 6.

Comments The Grid component Model is a hierarchical component model aimed specifically at grid systems. The reference implementation ProActive/GCM supports a wide range of features like components as units of concurrency, asynchronous

[4] Work published in [13] is part of this thesis, and is presented in Chapter 6.

communications with futures, reflective features such as introspection and dynamic re-configuration, etc. Consequently, we build our component model as a subset of full GCM specification. In [13] we build a mechanised model of GCM-like components along with runtime semantics so that we can reason on the execution of component applications and their evolution. Additionally, in [14], we extend the semantics presented in [1] to incorporate the formalisation of mechanisms for transmitting the values of futures to the objects/components that require them. Such a formalisation allows us to reason and prove properties of interest on these future update mechanisms.

2.6 Summary of Related Works and Positioning

In this chapter we presented an overview of the various languages/approaches that use futures as basic building blocks for distributed concurrent programming. We highlighted how these approaches handle creation, access and resolution of futures. We then proceeded to present how component based approaches have been adapted to distributed computing, what features are offered by each component model, and whether formal semantics are available for the component model or not. With an understanding of the general research domain of futures and component models for grid computing, we now use the following paragraphs to position our work relatively to what we have already presented. We summarise the different works presented in the preceding sections, and establish how the work presented in this thesis differs from the work done previously, and where it stands in the big picture of concurrent programming for distributed and grid systems.

Futures Futures are language constructs that improve concurrency in a natural and transparent way. First introduced in Multilisp and ABCL/1, futures provide an easy to program model for concurrent distributed applications; they provide data-flow synchronisation points. Futures may be explicit or implicit. Frameworks that make use of explicit constructs for creating futures include Multilisp [3], $\lambda(fut)$-calculus [5] , Creol [24, 6] , Java concurrent API, SafeFuture API [39], and ABCL/f [20]. In contrast, futures are created implicitly in frameworks like ASP [7], AmbientTalk [8, 22], ProActive [7, 15], and ASP_{fun} [51, 52]. In those object-oriented languages, implicit creation corresponds to an asynchronous method invocation. A key benefit of the implicit creation is that no distinction is made between synchronous and asynchronous operations in the program. Additionally, the futures can be accessed explicitly or implicitly. In case of explicit access, operations like `claim`, `get`, and `touch` are used to access the futures. For implicit access, operations that need the real value of an object, referred to as *blocking* operations, automatically trigger synchronisation with the future value transmission mechanism.

Alice ML [34, 21] and its formalisation $\lambda(fut)$ [5] provide futures that are created explicitly, but are resolved implicitly. Access to an unresolved future is a blocking operation until the value becomes available. In addition to futures, Alice ML and

$\lambda(fut)$ also provide a *Promise* [37] construct (handled futures in $\lambda(fut)$). A promise is an explicit handle for a future, and must be explicitly full-filled. Promise differs from regular future in the sense that a promise may be full-filled by any thread, with any future value. This differs from some other models where future value can only be computed strictly once, and only by the concurrent thread/process associated with the future.

Creol allows explicit control over data-flow synchronisations. In [47] Creol has been extended to support first class futures, although the future access is explicit. In [47], the authors provide the semantics of an object-oriented language based on Creol [6]; it features active objects, asynchronous method calls, and futures. They provide a proof system for proving properties relating to concurrency. The model is multi-threaded, with only one thread active at a given time. Our approach is quite close to this work except that we study a component model featuring high level of abstraction, and hierarchical composition. We do not provide a tool to prove properties on specific programs; we rather prove properties on the language/framework itself.

ASP [7] is an imperative distributed calculus with asynchronous communication between concurrent activities, no shared memory, and transparent implicit first class futures. Thus, the synchronisation is transparent and data-flow oriented. ProActive may be considered as one possible implementation of ASP calculus.

In AmbientTalk [8, 22], futures are also first-class and are transparently manipulated; but the future access is a non-blocking operation: it is an asynchronous call that returns another future. This avoids the possibility of a dead lock as there is no synchronisation. However, due to lack of any formal semantics for AmbientTalk, this property cannot be formally proved. This asynchronous access to futures, differs from the approach adopted in other frameworks like Alice ML, Creol, ASP, etc., where access to a future is blocking.

Futures in Java and other related works like Safe Future API, etc., are created and manipulated explicitly. These presented works aim to solve some of the problems that arise from access to shared memory resources.

Also in the context of object-oriented languages, ASP_{fun} [51, 52] is closely related to our work. $\text{ASP}_{\text{fun}}$is a complimentary calculus for ASP and deals with the functional fragment of ASP. The calculus is functional in the sense that all functions are applied on copies of arguments and thus there are no side-effects. It formalises a functional language featuring active objects, asynchronous communication, first class futures, and a type system. While the language provides for first class futures, it does not study future update strategies. Additionally, it does not deal with components.

Components Component modelling is a vast domain of active research, comprising approaches that vary from applied semi-formal approaches to formal methods. A number of models have been proposed both by the industry and by academia for development of component based applications.

COM and its distributed extension DCOM was developed by Microsoft for component-based development. Unlike other standards, COM and DCOM are binary standards and place no limits on how the components should be structured or implemented. Components communicate with outside world through well-defined immutable interfaces. The component models and their extensions in the .Net platform, used widely on Microsoft windows platform, are not a popular choice for large scale distributed systems.

The Enterprise Java Beans is a programming model developed by Sun Microsystems (now Sun-Oracle) for modelling for server-side components. It facilitates multi-tier programming paradigm and allows the programmer to focus on business code while leaving the features like persistence, fault tolerance, life-cycle management, transaction support, etc., to the containers which implement the specification. EJB is more suitable for large scale transactional enterprise systems and is closely tied with the java language. Additionally, while java does provide some reflection capabilities, runtime dynamic re-configuration of EJB is not possible.

The CORBA Component Model or CCM [58] is a standard defined by Object Management Group (OMG) [59]. The CCM specification supports business components which are distributed, heterogeneous and are independent of programming language and the underlying operating system. GridCCM [60] is an extension of CCM and provides components that can encapsulate SPMD style parallel programming model. However, both CCM and GridCCM do not provide any support for reflective features which are important for capabilities such as dynamic component reconfiguration.

A number of component models have been proposed specifically for distributed computing and computational grids. However, some well-known component models like CCA are not hierarchical – their intent is the efficient building, connecting and running of components but they neglect structural aspects. We rather focus on hierarchical component models like Fractal, GCM, or SCA.

SCA (Service Component Architecture) [69] is a component model adapted to Service Oriented Architectures. It enables modelling service composition and creation of service components.

SOFA 2 is another academic component model which supports component hierarchies, and provides runtime reflective capabilities for introspection, monitoring and dynamic re-configuration of components. The proposed model bears close resemblance to Fractal component model.

The Fractal component model [81, 71], is a modular and extensible component model. Fractal allows for hierarchical composition of components, and separation of functional and non-functional concerns. Fractal can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications. Fractal specification has been formalised in Alloy specification language [82]. The formalisation focuses on verifying the consistency of the model. In Comparison we consider asynchronous components and focus on the dynamic component behaviour. This is crucial when specifying future management procedures.

FraSCAti [72] is an implementation of the SCA model built upon Fractal, making this implementation close to GCM. It provides dynamic reconfiguration of SCA component assemblies, a binding factory, a transaction service, and a deployment engine of autonomous SCA architecture. Due to the similarity between FraSCAti and GCM, our approach provides a good approximation of formalisation of FraSCAti implementation.

Creol also offers components; the paper [44] presents a framework for component description and test. A simple specification language over communication labels is used to enable the expression of the behaviour of a component as a set of traces at the interfaces. Creol's component model does not support hierarchical structure of components.

Positioning As already discussed, futures provide an efficient and easy to use programming paradigm for distributed applications. In ASP, ProActive, Creol , etc., futures are first class entities; future references can be safely transmitted between processes. As references to futures disseminate, mechanisms are needed to keep track of future references and to propagate the computed result of each future to the processes that need it. Unlike other frameworks and calculi, ASP explicitly takes into account the possibility of using different future update mechanisms/strategies. Moreover, ProActive only provides implementation of one such strategy.

Our work focuses on efficient transmission of future values. We study various strategies that can be used for fetching the values for first class futures. We study both *eager* and *lazy* fetching of future value; focusing on efficiency of these strategies. On the implementation side, our work is based on the discussion of future update mechanisms presented in [7]. We extend the work presented in [16], not only by improving and extending the implementation to add support for nested-futures; we also provide a language independent semi-formal notation for modelling future update mechanisms. This work has been published in [10], and is presented in Chapter 3 and Chapter 4.

On the theoretical side, our work provides formalisation of hierarchical components and their structure. At our level of abstraction, this structure is shared by several component levels like Fractal, GCM, and SCA. However most implementations of SCA (except FraSCAti) do not instantiate the component structure at runtime. By contrast, to allow component introspection and reconfiguration at runtime, we consider a specification where structural information is still available at runtime. This enables adaptive and autonomic component behaviours. Indeed, component adaptation in those models can be expressed by reconfiguration of the component structure[91]. For example, reconfiguration allows replacement of an existing component by a new one, which is impossible or very difficult to handle in a model where component structure disappears at runtime.

Most existing works on formal methods for components focus on the support for application development whereas we focus on the support for the design and implementation of component models themselves. To our knowledge, our work is

the only one to support the design of component models in a theorem prover. It allows proving very generic and varying properties ranging from structural aspects to component semantics and component adaptation.

Our work extends [1] which presents a component model giving a semantics to GCM, including hierarchical components, asynchronous communication, and first class futures. In order to prove properties related to the implementation of futures, we have extended the runtime semantics presented in [1]; we present a runtime semantics for our components that includes formalisation of one future update protocol. With mechanised proofs, we show that our formalisation is complete and enables proofs on properties on futures and their update strategies, thus ensuring correctness of the ProActive/GCM implementation. This work is published in [14]. Other possible strategies are discussed in a semi-formal manner in [9, 10], along with a experimental evaluation of their efficiency.

Based on the experience gained in specification and proof demonstrated in [1, 14], we design a framework for supporting mechanised proofs for distributed components which is presented in [13]; in particular focusing on the handling of component structure, on a basic set of lemmas providing valuable tooling for further proof, and the illustration of the presented framework to prove a few properties dealing with component semantics and reconfiguration. With mechanised proofs, we show that our formalisation is complete and enables proofs on properties on futures and their update strategies, thus ensuring correctness of the ProActive/GCM implementation. Chapter 5 and Chapter 6 detail the theoretical aspects of this thesis.

Part I

Future Update Strategies: Specification and Implementation

Chapter 3

First Class Futures: Specification of Update Strategies

Contents

3.1 Background: Futures in ASP-Calculus 45

3.2 Background: Update Strategies for Futures 47

3.2.1 Classification of Future Update strategies 47

3.2.2 Eager Forward-based Strategy 48

3.2.3 Eager Message-based Strategy 49

3.2.4 Lazy Message-based Strategy 51

3.3 Semi-Formal Specification of Update Strategies 52

3.3.1 General Notation 52

3.3.2 Eager Forward-based Strategy 55

3.3.3 Eager Message-based Strategy 56

3.3.4 Lazy Message-based Strategy 58

3.4 Analysis of Future Update Strategies 59

3.5 Remarks on Semi-formal Specification of Strategies 62

Futures are language constructs that improve concurrency in a natural and transparent way (Chapter 2). A *future* is used as a place holder for the result of a concurrent computation [3, 4]. Once the computation is complete and a result (called *future value*) is available, the placeholder is *resolved*, i.e, placeholder is replaced by the result. Some frameworks and languages allow futures to be passed to other processes (local or remote) without requiring the actual value. We call such futures *first class futures* [7]. First class futures offer great flexibility in application design and can potentially improve concurrency both in object-oriented and procedural paradigms like work-flows [92, 93]. In case of first class futures, as futures propagate through the system, additional protocols are required to transfer result values when they are computed; the result value for a particular future should be transmitted to all processes which received that future. We refer to such protocols as *future update strategies*, and to the process of replacing a future by the result value computed for it, as a *future update*. It should be noted that the newly produced result value may itself contain other futures. We further discuss such nesting of futures while presenting future update strategies.

We present our work on specification of future update strategies. A high level specification of these future update strategies appeared previously in [7]. However, the specification presented there is abstract, and lacks any details on the working of the shown strategies or their underlying data structures. Our work on specifying future updates can be considered as an extension of [7]. However in contrast to [7], our specification is more precise and is detailed enough to be used – and has been used – as basis for a real implementation, as discussed in Chapter 4. We specify the future updates using a language independent approach that makes our work applicable to various existing frameworks like Creol, AmbientTalk, etc., that support first class futures. While we specify three main future update strategies, we believe that our approach is generic and is flexible enough to be adapted to other update strategies that can be envisioned. We aim to study the efficiency of the future updates; we use a simple model for analysing the costs involved in each of the strategies in terms of *number of message exchanges*, and *the time to update futures*. We hope this facilitate better understanding of future update strategies and justify the need to study such mechanisms in detail.

The three main contributions of this chapter (published in [9, 10]) are:

A Generic semi-formal notation

We introduce a generic and language independent notation for modelling the future update protocols. We specify the future update strategies in a *event-like* notation, with operations invoked by the strategies and events that are triggered by the middleware.

Semi-formal specification of update protocols

We use our generic semi-formal notation to specify three future update strate-

gies; two eager strategies (transmit results as soon as possible) and one lazy strategy (transmit results only on-demand) are shown.

Analysis of future update strategies
We informally estimate the efficiency of presented strategies in terms of *message exchanges* and *time to update futures*. We show a basic cost-analysis model for our selected strategies to better understand the trade-offs required by each strategy.

We base our view of futures on ASP-calculus . Therefore, we start the chapter with a brief look at futures in ASP-calculus; Section 3.1 summarises the utilisation and implementation of first class futures in ASP. Section 3.2 provides an informal view of future update strategies and illustrate their working using a sample scenario. A semi-formal specification of future update strategies then appear in Section 3.3 ; a general notation is presented followed by specification of three future update strategies. An analysis of various update strategies using a simple cost-model appears in Section 3.4. Finlay, Section 3.5 provides the concluding remarks on our specification and the presented analysis.

3.1 Background: Futures in ASP-Calculus

As previously discussed in Chapter 2, futures are temporary placeholder objects that represent the yet-to-be computed results of concurrent executions. Section 2.4.3 presented first class futures in ASP-calculus. To summarise, in ASP the basic unit of concurrency is an *activity*. An activity in ASP is analogous to a traditional process with its own execution thread. Each process has an associated *message queue* which stores incoming requests. All communications between processes are in the form of asynchronous method invocations—which we refer to as *requests*, and *replies*.

The asynchronous communication between processes is achieved through futures. When one process invokes an asynchronous method call on another process, the method call is wrapped as a request, and is enqueued at the invokee –the target process. The invoker receives a future as placeholder for the result of this request. Once the future is received, the invoker can continue its execution without waiting for the result. The processes dequeue requests from their message queues, and serve them according to some policy, for example FIFO. Once a request is executed/served and a result is produced, it is communicated to the processes/activities holding that future; at the destination this result transparently replaces the future. We refer to this resolving of futures as a *future update*.

Finally, access to an unresolved future is a *blocking* operation and the accessing execution thread is blocked until the future is resolved/updated. In ASP this is referred as a *wait-by-necessity* condition, and is essentially a data flow synchronisation. *Wait-by-necessity* indicates an actual requirement of result values, and the blocked processes cannot continue until the value becomes available. Once an update arrives, they can continue their processing. Although, *strict* operations which

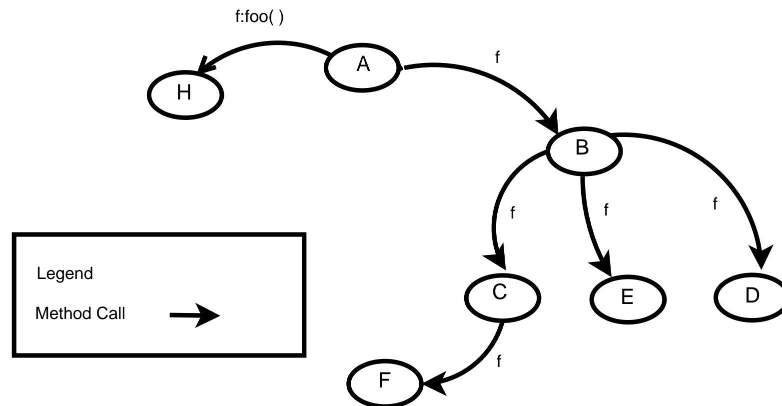

Figure 3.1: Futures propagate throughout the system

require access to the actual computed value for the future are blocking, it is possible to perform some operations on futures without blocking. Frameworks may choose to allow passing of futures as method arguments or return values without blocking, treating them as first class objects. Examples include ASP, AmbientTalk, etc.

Figure 3.1 illustrates how future references may spread through a system consisting of processes {A...F, H}. As the first step, process A makes an asynchronous invocation foo( ) on process H, obtaining the future f corresponding to yet-to-be computed result of foo( ). Future f is then communicated to process B from A as arguments of some asynchronous communication. B in turn forwards the future f to processes C, D, and E. Finally, future f arrives at process F through process C. It should be noted that each of the intermediate asynchronous invocations would result in futures as well, but for this example we focus only on the future f, ignoring all other futures.

At this stage, according to presented scenario, future f which corresponds to the request foo( ) enqueued at H, has propagated throughout the system. Once the request foo( ) is served (executed), and a result is available at H, it has to be transmitted to all processes interested in the value of future f.

First class futures increase the potential for parallelisms and are in fact a necessary requirement in some cases to avoid deadlocks. One such example, is the ProActive/GCM – the implementation of Grid Computing Model (GCM) in ProActive – where components can systematically deadlock in the absence of first class futures. The importance of first class futures in component models like ProActive/GCM is discussed in Section 5.2.5. A side effect of first class futures is that future references spread among various (local and remote) processes, necessitating additional mechanisms to ensure that results are communicated to all processes where they are needed/awaited. This is where future update strategies come into play. In the following section, we give an informal description of three main future update strategies, followed by a more precise treatment of future updates in Section 3.3.

3.2 Background: Update Strategies for Futures

It is possible to devise a wide variety of future update strategies for first class futures. However, in this thesis, we restrict our selves to the three update strategies which appeared previously in [7]. It should be noted that although we only present three future update strategies, we believe that our notation presented in Section 3.3 is language and framework independent, and is flexible enough to capture a wider variety of future update strategies that may be envisaged; this includes hybrid strategies that may arise from mixing of the strategies presented here.

3.2.1 Classification of Future Update strategies

We tag our future update mechanisms based on the answers to three questions: *When are futures updated?*, *which processes receive the results?*, and *which process communicates the results?*. The timing of future updates is directly linked to which processes get the results. *Eager* strategies aim at making the results available to processes as soon as possible; the result of a future is transmitted when it becomes available. This in-turn leads to transmitting the result of a future to all processes that received that particular future. On the other hand, *Lazy* strategies adopt a different approach and aim to reduce the number of future updates. Consequently, results are only sent to select processes. An alternative way of denoting strategies, can be based on where does the responsibility of transmitting the results lie; leading to *message-based* or *forward-based* strategies. In this thesis, we use *eager* and *lazy* as two broad categories, while each strategy is denoted with *message-based* or *forward-based* to provide further information on its working. To summarise, we classify future update strategies as:

Eager strategies: Strategies are called *eager* when the result of a given future is transmitted as soon as the future value is computed, i.e., the execution of the request associated with that future terminates. The goal of eager strategies is to minimise the time spent waiting for the results to arrive, once the results have been computed. However, eager strategies may increase the number of communications required, as a future value is proactively transmitted to all processes which received that future.

Lazy strategies: Strategies are called *lazy* if the results are only transmitted upon need, i.e., only when a process asks for that value. Lazy strategies try to minimise the number of future updates, by restricting who may receive the future value. This policy of only transmitting the results on-demand may potentially reduce number of communications transferring results. However, this requirement of explicit demand may increase the time spent waiting for a future value. Any process that requires the value for a given future must ask for it, thus introducing an additional delay (if the result is already computed).

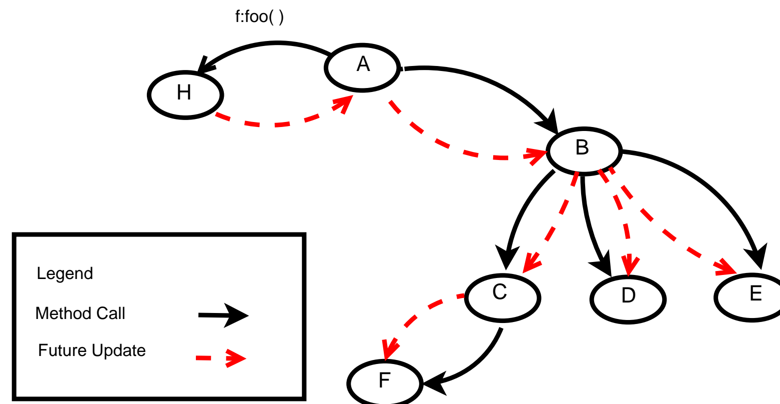

Figure 3.2: Eager forward-based: Future updates follow the flow of futures

In the following subsections, we describe two eager strategies and one lazy strategy: *eager forward-based* (follow the flow of futures), *eager message-based*(register all future receivers), and *lazy message-based* (only register on wait-by-necessity). One could also consider a lazy forward-based strategy, but as it is extremely inefficient, we do not discuss it here.

3.2.2 Eager Forward-based Strategy

In eager forward-based strategy, future updates follow the same path as taken by the flow of the corresponding future. Each process *remembers* the pair –destination process and the future– whenever it communicates (forwards) a future to another process. In a real implementation, this information may be stored in any data-structure capable of storing mappings. When a result becomes available for a particular future at any given process, this memorised information is used to determine where (to which processes) the result should be sent. As a consequence, starting from the source computing the future value, the future updates follow the same path as the future before it. Each process receives the result, performs a local update (resolves the future locally) and then forwards the result to the next process in the chain, the processes to which it previously forwarded the future. The process is repeated at each incremental process in the chain, until all processes that received the future, also receive the update value.

Figure 3.2 shows the working of eager forward-based strategy using scenario introduced in Figure 3.1. The method invocations, and consequently the flow of future f is shown by continuous (black) lines. Once the request `foo( )` is served and a result computed, process `H` sends the result to process `A` as shown by a dashed (red) line, where future f is updated with the result. As discussed above, `A` 'remembers' that future f was forwarded to `B`. Therefore, `A` sends the received result to `B`. Similarly, in its turn, `B` remembers that the result for future f should be sent to to `C`, `D` and `E` and communicates the result to them. Finally, f is updated at process `F` with result that arrives via process `C`.

Comments Eager forward-based strategy is available as the default future update strategy in ProActive distributed programming library [15]. As the name implies, It is an eager strategy and results are updated as soon as they are available. Result for a given future is sent to all the processes that received that future. In eager forward-based strategy, the responsibility of updating a future does not rest with any one particular process. Each process is responsible for communicating the result to the processes to which it sent the future. This allows future updates to propagate inside the system in a de-centralised manner. At each process, the only information required is *from where a particular future was received* ?, and *where it was forwarded* ? A process needs not know for example, which process will compute the value for a future. This allows a future to be only loosely coupled with the process that will compute its value. This is particularly useful when implementing features such as process migration. However, process migration is not within the scope of this thesis and hence is not further discussed here. Another useful consequence of this loose coupling is easy garbage collection of computed and received results. Once a process has sent the results to all processes to which it forwarded the future, the result is not longer needed and may be safely garbage collected.

Finally, as a consequence of following the same path as flow of futures, the wait-time for a particular result varies among processes and depends on their *distance* – number of intermediate processes – from the source process computing the result. A result value may have to pass through a number of intermediate processes before arriving at a given process.

3.2.3 Eager Message-based Strategy

In eager message-based strategy, the process computing the future value directly communicates the result to all the processes which received that future. In contrast to forward-based strategy where future updates are performed in a distributed manner (albeit along process-chains), in eager message-based strategy all updates are communicated by the process computing the value; the values are transmitted in a centralised manner. For such a scheme to work, a mechanism is needed to inform the computing process where the results should be sent. This is achieved using a registration message; each time a future is forwarded, every process which receives the future is registered with the computing process. This registration message may be sent by either the process forwarding the future or the one that receives it, depending on the particular implementation/policy. Additionally, each future carries the information about which process will compute its value to facilitate registrations. This information is easily available and is embedded in the futures when they are created (recall that futures are created in response to a asynchronous invocation on a process). The strategy is eager, as all processes which receive a future reference are registered and hence they all receive the future value once it becomes available.

Figure 3.3 shows the future updates and registration in eager message-based strategy. The method invocations and hence the flow of futures is shown by continuous (black) lines as before. However, in case of eager message-based strategy, in

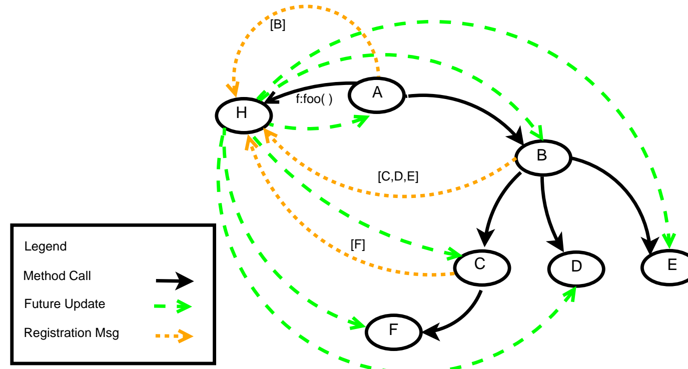

Figure 3.3: Eager message-based: All future recipients register

addition to method invocations, we have dotted (brown) lines showing the registration messages. As mentioned before, the registration message can be sent either by the receiver or the sender process. For convenience, we show a registration message being sent by the process forwarding the future, registering all the processes which receive the future. The use of registration message to select which processes receive the results, leads to the strategy being denoted as message-based.

Therefore, when process `A` forwards the future f to process `B`, it registers `B` as a future recipients with process `H`. Similarly, `B` sends a registration message to `H`, registering processes `C`, `D` and `E` as future recipients. Finally, `C` forwards the future to `F` and registers it with `H`. Once `H` finishes the execution of `foo( )`, and produces a result, `H` is responsible for communicating the result to all registered processes (effectively all processes that received the future f). Therefore `H` sends the value directly to processes {`A`...`F`} (shown in green).

Comments Eager message-based strategy is an eager strategy. The result for a given future is transmitted to all process which received the future, as soon as the result becomes available. All updates are performed by the process which computes the result value in a centralised manner. Opposed to eager forward-based strategy, the result arrives at all processes in a single step, without going through intermediate processes. As a result, future updates arrive at all processes within constant time and only depend on the number of concurrent communications that can be initiated at the computing process. However, this strategy requires more communications (message-exchanges) than eager forward-based strategy; additional registration messages are used to keep track of future recipients. Also, futures are more tightly coupled with the processes computing their values to facilitate registration. Each process can look at the future and discover where (to which process) to send the registration message. Such tight coupling makes support for features such as process migration more complex.

The garbage collection for eager message-based strategy also require greater synchronisation. The computing process needs to ensure that all registered processes

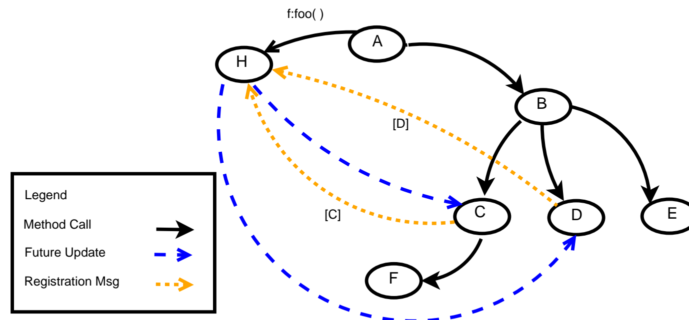

Figure 3.4: Lazy message-based: Register only on wait-by-necessity

have received the future value, and that there are no more futures corresponding to this value in transit. Once this is achieved, the computed values may be garbage collected. This is further discussed in Chapter 9 (future works). Finally, as all the updates are being done centrally by one process, it may consume more bandwidth for future updates than eager forward-based strategy.

3.2.4 Lazy Message-based Strategy

Lazy message-based strategy tries to minimise the number of future updates by delaying the updates until the future value is actually required. The strategy closely resembles the eager message-based strategy, but unlike eager message-based strategy, forwarding a future does not trigger a registration message. Consequently, futures are forwarded by a process without registering the processes which receive the future – the future recipients. Registration messages are triggered only in response to a wait-by-necessity condition, i.e, when a process tries to access the value of a unresolved future and is blocked. Thus only the processes which require the actual value and are blocked waiting for the result, are registered. Only those registered processes receive the future value. The remaining processes which received the future, but make no attempt to access the future (and are not blocked as a result), do not receive the value when it is computed. This may potentially reduce the number of processes to which future value should be sent, thus saving bandwidth. The computed results are stored, in case some processes ask for them later on.

Figure 3.4 shows the working of lazy message-based strategy. In contrast to eager message-based strategy no registration is performed when futures are forwarded. Therefore, no registration needs to be performed by processes `A`, `B` and `C` while forwarding the future f. Registrations are performed only when there is a wait-by-necessity. Suppose in our example, `C` and `D` attempt to access the unresolved future f and are consequently blocked. The resulting wait-by-necessity triggers the registration of `C` and `D` with `H` for the value of future f. When this value is computed, `H` sends it directly to `C` and `D` (shown in blue). No other process receives the future value.

Comments The lazy message-based strategy closely resembles the eager message-based strategy without the mandatory registration of all processes which receive the future. Processes are are registered only in response to a wait-by-necessity condition, signalling that the process actually requires the future value. The lazy registration of processes may potentially reduce the number of updates performed. This can result in significant savings in the number of communications where only a few processes actually use the result value. On the other hand, the lazy registration introduces an additional time delay before the result arrives where they are needed.

In lazy strategy only the processes which demand a result value, receive it. Consequently, the computed values are stored, as some processes may ask for the value later on. As a result, the garbage collection of computed values for lazy message-based strategy is more complex than previous strategies. In general, a result value may not be garbage collected in a lazy strategy, unless extra measures are taken to ensure that no further process will ask for the value. Alternatively, a result value *time-out* style may be adopted, depending on the nature of the applications.

Finally, the lazy message-strategy shines in situations where only a minority of processes trigger wait-by-necessity. In cases where all processes or majority of processes require result values, it may perform worse than eager message-strategy due to additional delays caused by lazy registration.

3.3 Semi-Formal Specification of Update Strategies

This section presents a semi-formal notation to model the future update strategies. We use the presented notation to specify the three update strategies discussed in the previous sections. Two eager and one lazy strategies are presented here: *eager forward-based strategy*, *eager message-based strategy*, and *lazy message-based strategy.*

3.3.1 General Notation

This section presents a brief overview of the various notations and constructs that we use to model the future update strategies. We denote by $\mathcal{A}$ the set of processes (*activities*); $\alpha, \beta, \ldots \in \mathcal{A}$ range over processes. $\mathcal{F}$ denotes the set of futures, each future is of the form $f^{\alpha\rightarrow\beta}$, which represents the future f created by the process α, that will be computed by process β. The future f is created as a placeholder for the result of asynchronous method invocation by process α on process β. As each process needs to keep track of the futures it has received, we make use of some local lists for this purpose. There is one *future list* for each process in $\mathcal{A}$. It represents the location where the futures are stored in local memory. For a process α:

$$\mathcal{FL}_\alpha \colon \mathcal{F} \mapsto \mathcal{P}(Loc)$$

Locations, called *loc* in the following and of type *Loc*, refer to the in-memory position of the future. To keep track of processes to which the result value for a given future should be sent, a *future recipient* list is maintained in each process.

$$\mathcal{FR}_\delta \colon \mathcal{F} \mapsto \mathcal{P}(\mathcal{A})$$

$\gamma \in \mathcal{FR}_\delta\,(f^{\alpha\to\beta})$ if the value for future $f^{\alpha\to\beta}$ has to be sent from process δ to process γ. It should be noted that each future $f^{\alpha\to\beta}$, can be mapped to several locations in $\mathcal{FL}$ or several processes in $\mathcal{FR}$. $\mathcal{FR}$ and $\mathcal{FL}$ are initialised to empty mappings on all processes. It should be noted that while both the $\mathcal{FR}$ and $\mathcal{FL}$ are mappings, for the most part we refer to them as lists for convenience.

We use an event-like notation to define the different strategies. Operations –invoked by the strategies– and events triggered by the middleware are described below respectively. Events and operations are indexed by the processes on which they occur, and are noted as $\alpha \to \beta$, for a communication (asynchronous message or a future update) from process α to process β.

Operations We define various operations that may be invoked by the future update strategies. These are generic operations and are useful regardless of the exact strategy being used; defining them separately helps in simplifying specification of individual strategies. We show these operations in the following:

Register future - Reg: $\mathcal{F} \times \mathcal{B} \times (\mathcal{F} \mapsto \mathcal{P}(\mathcal{B}))$
We define an operation *register future* – *Reg*– that takes three arguments: a future, a process and a mapping $\mathcal{F} \mapsto \mathcal{P}(\mathcal{B})$ (either $\mathcal{FL}$ when $\mathcal{B} = Loc$, or $\mathcal{FR}$ when $\mathcal{B} = \mathcal{A}$). $Reg_\gamma(f^{\alpha\to\beta}, b, L)$ replaces the list (mapping) L by the list L' defined as follows:

$$L'(f_2^{\alpha'\to\beta'}) = \begin{cases} L(f^{\alpha\to\beta}) \cup \{b\} & \text{if } f_2^{\alpha'\to\beta'} = f^{\alpha\to\beta} \\ L(f_2^{\alpha'\to\beta'}) & \text{else} \end{cases}$$

The *Reg* operation replaces the old mapping L with a new mapping L', such that L' contains the mappings in L plus an additional mapping. An example usage could be $Reg_\gamma(f^{\alpha\to\beta}, \delta, \mathcal{FR}_\gamma)$. Here, Reg_γ, indicates that *Reg* is invoked on process γ. The invocation, adds in the future recipient list $\mathcal{FR}_\gamma$, a new process δ associated to future $f^{\alpha\to\beta}$; this new mapping indicates that future $f^{\alpha\to\beta}$ has been forwarded to process δ as well.

Locally update future with value - Update: $Loc \times Value$
Once the value for a given future is received, this operation is triggered to update all corresponding local futures with this value. The operation $Update_\gamma(loc, v)$ replaces, in the activity γ, a reference to a future $f^{\alpha\to\beta}$ by the value v. This reference is situated at location *loc*. Remember the set of locations of these references is $\mathcal{FL}_\gamma(f^{\alpha\to\beta})$.

Clear future from list - Clear: $\mathcal{F} \times (\mathcal{F} \mapsto \mathcal{P}(\mathcal{B}))$
The clear operation $Clear(f^{\alpha\to\beta}, L)$ removes the entry for future $f^{\alpha\to\beta}$ from the list

L; it replaces the list L by the list L' defined by:

$$L'(f_2^{\alpha' \to \beta'}) = \begin{cases} L(f_2^{\alpha' \to \beta'}) & \text{if } f_2^{\alpha' \to \beta'} \neq f^{\alpha \to \beta} \\ \emptyset & \text{else} \end{cases}$$

It will be used after a future update to clear entries for the updated future. This ensures that once a future is updated with a value, it no longer exists.

Send future value: SendValue: $\mathcal{F} \times Value$
Send operation is used when a process needs to send the computed result value of a future to another process; this value is used to update the futures there. An example usage could be $SendValue_{\delta \to \gamma}(f^{\alpha \to \beta}, v)$. *SendValue* operation is a communication from process δ to process γ and is thus marked as $\delta \to \gamma$. The operation communicates the result value v for the future $f^{\alpha \to \beta}$ from process δ to process γ. As discussed before, the result value may itself be partial, i.e., it may contain other futures. Consequently, sending a future value can trigger *send future reference* events – *SendRef* – for all the futures inside the value v. The details of this operation appear in Sections 3.3.2, 3.3.3, and 3.3.4.

Events

Future update strategies react to events, triggered by the application or the middleware. We present these events below:

Create future: Create: $\mathcal{F} \times Loc$
$Create_\alpha(f^{\alpha \to \beta}, loc)$ is triggered when process α makes an asynchronous invocation on process β. Process α creates the future $f^{\alpha \to \beta}$ corresponding to this invocation. The result of $f^{\alpha \to \beta}$ will be computed by the process β. The semantics of this event is similar for all strategies: it registers the future in the future list $\mathcal{FL}$ of the creating process.

$$Create_\alpha(f^{\alpha \to \beta}, loc) \triangleq Reg_\alpha(f^{\alpha \to \beta}, loc, \mathcal{FL}_\alpha)$$

Send future reference: SendRef: $\mathcal{F} \times Loc$
$SendRef_{\delta \to \gamma}(f^{\alpha \to \beta}, loc)$ occurs when the process δ sends the future reference $f^{\alpha \to \beta}$ to γ and the future is stored at the location *loc* on the receiver side. The details of this operation will be described in Sections 3.3.2, 3.3.3, and 3.3.4.

Future computed: FutureComputed: $\mathcal{F} \times Value$
$FutureComputed_\beta(f^{\alpha \to \beta}, v)$ denotes the *end of request service*(execution). This event occurs when process β completes the execution of request corresponding to future $f^{\alpha \to \beta}$, producing the result value v. The reaction to this event is described later on for each individual strategy.

Wait-by-necessity: Wait: $\mathcal{A}$
This event is triggered when a process accesses an unresolved future. This corresponds to *get* or *touch* operation in [6, 24, 20]. For the two eager strategies it simply causes the process to be blocked until the value is received. For the lazy strategy, this event retrieves the future value, see Section 3.3.4.

3.3.2 Eager Forward-based Strategy

We presented the eager forward-based strategy in Section 3.2.2. To summarise, in this strategy, each process remembers the processes to which it forwarded the future. When the result value is available at a process, it is sent to all processes to which the future was forwarded previously. Thus the result value in the future update follow the same path as the flow of the future corresponding to this result. When the result value for a future $f^{\alpha\to\beta}$ is available at process γ, the list of processes to which this value should be forwarded, is given by $\mathcal{FR}_\gamma(f^{\alpha\to\beta})$. It is the list of processes to which γ had previously sent the future $f^{\alpha\to\beta}$. In case of $\gamma = \beta$ (computing process), the result is sent to process α which created this future (future $f^{\alpha\to\beta}$ is the placeholder for result of asynchronous invocation on process β by process α).

Figure 3.5 (reproduced from Figure 3.2) shows an example illustrating this strategy. In the presented scenario, process A makes an asynchronous call on process H and receives (creates) the future $f^{A\to H}$. Process A then passes this future to process B, which in turn passes the future to processes C, D and E. Finally C passes the future to F. Each time a future is forwarded, i.e., a request/reply containing a future is communicated – modelled by *SendRef* communication – the forwarding process δ adds the destination to its $\mathcal{FR}_\delta(f^{A\to H})$. Thus, in the illustrated example, when the future $f^{A\to H}$ is forwarded from process A to process B, A adds B to $\mathcal{FR}_A(f^{A\to H})$. When process H finishes the execution of request and produces a result value v for the future $f^{A\to H}$, it is communicated to A using a *SendValue* message. Process A then forwards the update to process B ($\mathcal{FR}_A(f^{A\to H}) = \{B\}$). Similarly, at B ($\mathcal{FR}_B(f^{A\to H}) = \{C, D, E\}$) the future update value should be sent to C,D and E. Process B can communicate the value concurrently to those processes. Finally, the process F is updated after receiving the future update from process C ($\mathcal{FR}_C(f^{A\to H}) = \{F\}$).

Send future reference When a process δ sends a future $f^{\alpha\to\beta}$ to a process γ, the sender registers the destination process in $\mathcal{FR}_\delta$, and the destination process registers the location of the future in $\mathcal{FL}_\gamma$.

$$\mathit{SendRef}_{\delta\to\gamma}(f^{\alpha\to\beta}, loc) \triangleq \;\; \mathit{Reg}_\delta(f^{\alpha\to\beta}, \gamma, \mathcal{FR}_\delta); \;\;\; \mathit{Reg}_\gamma(f^{\alpha\to\beta}, loc, \mathcal{FL}_\gamma)$$

Future computed Once the value of a future $f^{\alpha\to\beta}$ has been computed at process β, it is immediately sent to all the processes that belong to $\mathcal{FR}_\beta(f^{\alpha\to\beta})$. This will trigger chains of *SendValue* operations. Once the future value have been sent, the

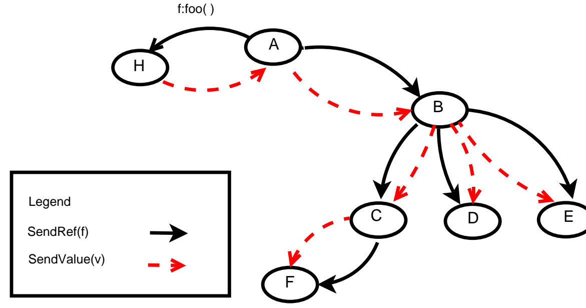

Figure 3.5: Future-update in eager forward-based strategy

entry in the future recipient list is no longer useful and can be removed:

$$\begin{aligned} FutureComputed_{\beta}(f^{\alpha\to\beta}, value) \triangleq\ & \forall\, \delta \in \mathcal{FR}_{\beta}(f^{\alpha\to\beta})\ SendValue_{\beta\to\delta}(f^{\alpha\to\beta}, value); \\ & Clear_{\beta}(f^{\alpha\to\beta}, \mathcal{FR}_{\beta}) \end{aligned}$$

Send future value When a future value is received, the receiver first updates all the local occurrences (the *loc*(s) in its $\mathcal{FL}$ list), and then sends the future value to all the processes to which it had forwarded the future (the processes in its $\mathcal{FR}$ list). The operation is recursive, because the destination process of *SendValue* may also need to update further futures. This operation can potentially trigger the *SendRef* operation in case of nested futures (value can itself contain futures). The future locations and entries in future recipient list for this future are not needed anymore after those steps:

$$\begin{aligned} SendValue_{\delta\to\varepsilon}(f^{\alpha\to\beta}, value) \triangleq\ & \forall\, loc \in \mathcal{FL}_{\varepsilon}(f^{\alpha\to\beta})\ \ Update_{\varepsilon}(loc, value); \\ & Clear_{\varepsilon}(f^{\alpha\to\beta}, \mathcal{FL}_{\varepsilon}); \\ & \forall\, \gamma \in \mathcal{FR}_{\varepsilon}(f^{\alpha\to\beta})\ \ SendValue_{\varepsilon\to\gamma}(f^{\alpha\to\beta}, value); \\ & Clear_{\varepsilon}(f^{\alpha\to\beta}, \mathcal{FR}_{\varepsilon}) \end{aligned}$$

3.3.3 Eager Message-based Strategy

We presented the eager message-based strategy in Section 3.2.3. To summarise, in eager message-based strategy, the process computing the value of a future, is responsible for updating all processes which previously received that future. Opposed to forward-based strategy where futures updates are performed in a distributed manner, here all updates for a given future are performed by the same process in a centralised manner. Therefore, for a future $f^{\alpha\to\beta}$, created due to an asynchronous call from process α to process β, all updates are done by process β when the execution of request terminates and a result value becomes available. To keep track of where to send the results, every process receiving the future $f^{\alpha\to\beta}$ is registered with β. As already discussed, this registration message can be sent either by the process forwarding the future, or by the process receiving the future. In general, whenever a process δ forwards a future to another process γ, a *registration message* is sent to

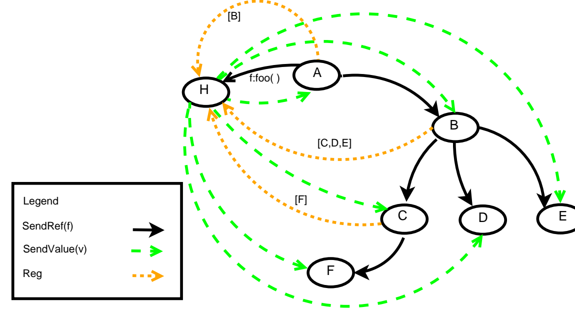

Figure 3.6: Future-update in eager message-based strategy

the process β (responsible for computing the result), registering γ as a future recipient in $\mathcal{FR}_\beta$. Recall that $\mathcal{FR}_\beta(f^{\alpha\to\beta})$ contains the list of processes to which $f^{\alpha\to\beta}$ has been forwarded. The registration messages is a communication informing the computing process about future recipients. To simplify our notation we represent the *registration message* by invoking the previously seen *Reg* operation remotely at the target process (identified by the subscript).

Figure 3.6 (reproduced from Figure 3.3) shows an example of this strategy. When the process A forwards the future f to the process B, B should be registered in $\mathcal{FR}_\mathcal{H}$. Therefore, the operation $Reg_H(f^{\alpha\to\beta}, B, \mathcal{FR}_\mathcal{H})$ is invoked remotely on process H, registering B in $\mathcal{FR}_H$. Similarly we have registrations invoked on H from B adding C, D, and E to $\mathcal{FR}_H$; finally we have $\mathcal{FR}_H(f^{A\to H}) = \{A, B, C, D, E, F\}$. Once the future result is available, H uses the *SendValue* message (shown in green) to communicate the value to all processes in $\mathcal{FR}_H(f^{A\to H})$.

Send future reference In the message-based strategy when a future $f^{\alpha\to\beta}$ is forwarded by a process δ to a process γ, the process γ should be registered in future recipient list ($\mathcal{FR}_\beta$) of process β (β is the computing the result).

$$SendRef_{\delta\to\gamma}(f^{\alpha\to\beta}, \gamma, loc) \triangleq \; Reg_\beta(f^{\alpha\to\beta}, \gamma, \mathcal{FR}_\beta); \quad Reg_\gamma(f^{\alpha\to\beta}, loc, \mathcal{FL}_\gamma)$$

The registration is performed by using the operation $Reg_\beta(f^{\alpha\to\beta}, \gamma, \mathcal{FR}_\beta)$ remotely on process β. In Figure 3.6, this is indicated by the arrow *Reg*.

Future computed Once the execution of a request terminates, and the result value for future $f^{\alpha\to\beta}$ is produced in process β, it sends the value to all the processes registered in $\mathcal{FR}_\beta(f^{\alpha\to\beta})$. After the transmission of results, entries for $f^{\alpha\to\beta}$ in $\mathcal{FR}_\beta$ can be removed.

$$\begin{aligned} FutureComputed_\beta(f^{\alpha\to\beta}, value) \triangleq \; & \forall\, \delta \in \mathcal{FR}_\beta(f^{\alpha\to\beta}) \; SendValue_{\beta\to\delta}(f^{\alpha\to\beta}, value); \\ & Clear_\beta(f^{\alpha\to\beta}, \mathcal{FR}_\beta) \end{aligned}$$

Send future value Contrarily to forward-based strategy, there is no need to forward the future value when received, only local references are updated, and then

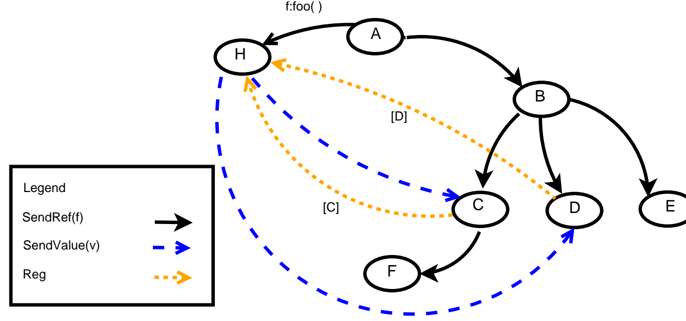

Figure 3.7: Future update in lazy message-based strategy

the $\mathcal{FL}$ list can be cleared.

$$SendValue_{\beta\rightarrow\gamma}(f^{\alpha\rightarrow\beta}, val) \triangleq \forall\, loc \in \mathcal{FL}_\gamma(f^{\alpha\rightarrow\beta})\ \ Update_\gamma(loc, val); \\ Clear_\gamma(f^{\alpha\rightarrow\beta}, \mathcal{FL}_\gamma)$$

The received future value may contain other futures as well. In this case, it can potentially trigger the send future reference operation, as discussed previously.

3.3.4 Lazy Message-based Strategy

We presented the lazy message-based future update strategy in Section 3.2.4. To summarise, the lazy strategy differs from the eager strategies in the sense that future values are only transmitted when absolutely required. When a process accesses an unresolved future, the access triggers the future update. This strategy is somewhat similar to message-based strategy except the futures are updated only when necessary. The process requiring the result value accesses the unresolved future, triggering the *wait-by-necessity* (*WBN*). Instead of simply blocking on *WBN* like other strategies, in lazy strategy registration is performed on *WBN*. The process then blocks until the result value arrives. As discussed previously, each process now needs to store all the future values that it has computed. For this, we introduce another list, $\mathcal{FV}$ that stores those values: $\mathcal{FV}\colon \mathcal{F} \mapsto \mathcal{P}(Value)$. $\mathcal{FV}_\beta(f^{\alpha\rightarrow\beta})$, if defined, contains a singleton, which is the result value for future $f^{\alpha\rightarrow\beta}$. Recall that the result itself may contain other futures.

Compared to eager message-based strategy, in lazy message-based strategy as shown in Figure 3.7 only the processes that require the future value register in $\mathcal{FR}_H$, $\mathcal{FR}_H(f^{A\rightarrow H}) = \{C, D\}$ if only C and D access the future, triggering *WBN*. When the result is available, H communicates it to processes in $\mathcal{FR}_H(f^{A\rightarrow H})$. In addition, the value is stored in $\mathcal{FV}_H(f^{A\rightarrow H})$. If the future value is required later, it will be retrieved from $\mathcal{FV}_H(f^{A\rightarrow H})$.

Send future reference This strategy does not require any registration with the process computing the future value, when a future is forwarded. When a future $f^{\alpha\rightarrow\beta}$ is forwarded by process δ to another process γ, the incoming future is registered

locally in $\mathcal{FL}_\gamma$ on the receiver-side (process γ). Once the value is received, all local references can be updated.

$$SendRef_{\delta\to\gamma}(f^{\alpha\to\beta}, \gamma, loc) \triangleq Reg_\gamma(f^{\alpha\to\beta}, loc, \mathcal{FL}_\gamma)$$

Wait-by necessity Wait-by-necessity is triggered when a process tries to access the value of an unresolved future. In case of an access to a future $f^{\alpha\to\beta}$ in process γ, we attempt to register the waiting process γ in the future recipient list $\mathcal{FR}$ of process β. If the future has already been computed by process β (result is available in $\mathcal{FV}_\beta$), the value is transmitted immediately. Otherwise, the registering-process is added to the future recipient list of process β.

$$\textit{Wait-by-necessity}_\gamma(f^{\alpha\to\beta}) \triangleq \begin{cases} SendValue_{\beta\to\gamma}(f^{\alpha\to\beta}, val) & \text{if } \mathcal{FV}_\beta(f^{\alpha\to\beta}) = \{val\} \\ Reg_\beta(f^{\alpha\to\beta}, \gamma, \mathcal{FR}_\beta) & \text{if } f^{\alpha\to\beta} \notin dom(\mathcal{FV}_\beta) \end{cases}$$

Future computed When a result is computed, the value is stored in the future value list. Moreover, if there are pending requests for the value (registrations in $\mathcal{FR}$), then the value is sent immediately to all the awaiting processes. Once the value has been transmitted, entries for that future can be removed from $\mathcal{FR}$. Entries in $\mathcal{FV}$ cannot be removed as discussed previously.

$$\begin{aligned} FutureComputed_\beta(f^{\alpha\to\beta}, val) \triangleq\ & \forall\, \delta \in \mathcal{FR}_\beta(f^{\alpha\to\beta})\ SendValue_{\beta\to\delta}(f^{\alpha\to\beta}, val) \\ & Clear_\beta(f^{\alpha\to\beta}, \mathcal{FR}_\beta);\ Reg_\beta(f^{\alpha\to\beta}, val, \mathcal{FV}_\beta) \end{aligned}$$

Send future value The *SendValue* operation for lazy message-based strategy is the same as for the eager message-based strategy:

$$\begin{aligned} SendValue_{\beta\to\gamma}(f^{\alpha\to\beta}, val) \triangleq\ & \forall\, loc \in \mathcal{FL}_\gamma(f^{\alpha\to\beta})\ Update_\gamma(loc, val); \\ & Clear_\gamma(f^{\alpha\to\beta}, \mathcal{FL}_\gamma) \end{aligned}$$

3.4 Analysis of Future Update Strategies

In this section, we present a simple model for analysing the cost of updating futures using different strategies. Using the detailed specification of future update strategies presented in the preceding sections, we *informally estimate* the efficiency of strategies. We base our estimation on *number of message exchanges* and *the time to update futures*. We hope this cost-analysis of our presented strategies will facilitate a better understanding of the the costs and trade-offs required by each strategy.

Assumptions and limitations For the purpose of our analysis, we assume that futures are forwarded over a simple *non-cyclic tree* like configuration of processes. The graph of the future updates is non-cyclic as a given future can be updated only once. Therefore, even when cycles are encountered, they have no impact on our analysis. The cost analysis focuses on the time necessary for updates and number of message exchanges required and does not consider the computation time which

is too application dependent. It is also equally difficult to estimate at what stage during the execution of a request, a process will try to access a given future leading to wait-by-necessity. Our intent is to keep the model as simple as possible, and as such, we do not take this random delay into account. We assume that all results have already been produced and start our analysis from there on. So for us:

In case of eager forward-based strategy, the processes are all assumed to have just started waiting for the future value. The computing process is assumed to have finished executing the request and has produced the result; and consequently is about to start future updates.

In case of eager message-based strategy, we assume that all the processes which received the future, have already registered with the process computing the result value. Those processes are now in wait state. The computing process has finished request execution and the value has just now become available.

In case of lazy message-based strategy, we assume that the processes which require the future-value have just made an attempt to access the value. Consequently, the resulting registration message have not been sent yet.

Notation and configuration Our analysis focuses on a tree T consisting of N processes; depth of tree T is given by the function $D(T)$. We assume that all processes have the same degree d. The degree of a process is *number of processes to which it has forwarded the future* $f^{\alpha\to\beta}$. If the degree is not constant, d can be approximated by the average degree of the processes in T, and the cost is approximated. At each process, the maximum number of concurrent updates a process can perform is given by k; in practise k is the size of thread pool. Additionally, for lazy strategy, only l processes out of the total N processes make use of future values. In order to update future values on various processes in the tree T, the result value must be serialised-deserialised at the appropriate processes. The time spent in serialising-deserialising for one transfer is denoted by t_s, while t_f is the time required for transmitting the serialised result. Additionally, t_r is the time for registering a process as a future recipient (for the lazy strategy). As discussed in assumptions, for eager strategies, the registrations are done at the time of future forwarding and as a results are not considered. Finally, we use the notation $\lceil .. \rceil$ for denoting the ceiling function on real numbers.

Using these notations, we aim to approximate: a) the total number of messages needed to update a given future $f^{\alpha\to\beta}$, b) The total time needed to update a given future $f^{\alpha\to\beta}$ at all N processes, c) T_w: the time for a given process γ to receive the result for future $f^{\alpha\to\beta}$.

Table 3.1 presents a summary of our cost estimation. In eager forward-based strategy the responsibility of future update is distributed among all intermediate processes. This can be an important consideration in environments where the bandwidth available is limited. On the other hand, this implies that future update time is dependent on the number of intermediate processes that must be traversed. Each intermediate process requires serialisation-deserialisation. As can be observed

Table 3.1: Summarised cost-model

Variable	**Eager forward-based**	**Eager message-based**	**Lazy message-based**
Number of messages	N	$2 * N$	$2 * l$
Time to update all futures	$D(T) * (t_f + t_s) * \lceil d/k \rceil$	$\lceil N/k \rceil * t_f + t_s$	Not Applicable
Time to update a future at a given process	$t_f + t_s \leq T_w \leq D(\gamma) * (t_f + t_s) * \lceil d/k \rceil$	$t_f + t_s \leq T_w \leq \lceil N/k \rceil * t_f + t_s$	$t_s + t_f + t_r \leq T_w \leq \lceil l/k \rceil * t_f + t_s + t_r$.

from Table 3.1, the time spent in serialisation-deserialisation t_s is an important factor along with the number of intermediate processes that must be *traversed* before results arrive at a particular process. As the depth of the tree increases (more intermediate hops), t_s can become significantly high, as compared to t_f, indicating a large overhead in time wasted on serialisation-deserialisation. The time it takes for a result to arrive at a given process γ depends on the position of the processes inside the tree $D(\gamma)$, and the overhead required for serialisation-deserialisation at each intermediate process. Finally, the number of message exchanges required to update all processes is the same as the number of processes which receive the future value. Therefore, for N processes receiving the future, N future update messages are generated.

In eager message-based strategy the responsibility to update the future values is centralised at computing process. This can potentially over-load the process and the available bandwidth. On the other hand, all updates are carried out by a single process, thus results need to be serialised only once if group communication mechanisms are employed. Also, the update time is independent from the location of the process, and all processes receive future update in a relatively constant time. This can be an important consideration in scenarios where the serialisation-deserialisation time t_s is relatively large and the depth of the tree is significant. The limiting factor in case of eager message-strategy is the number of concurrent updates that may be initiated by the computing process. The time to update processes simply becomes a function of the number of concurrent updates, as the serialisation is done only once, and t_s remains constant for all updates.

For lazy strategy, the registration requests can arrive at any time, before or after future value has been computed (although we only consider the case where registration request arrives after the computation). The main drawback of this strategy is that, since registration is performed only on wait-by-necessity, the access to a future necessarily waits for one registration request time t_r plus the time to update a future t_f. This introduces additional delay compared to the eager strategies; however, there may not be any additional delay if *WBN* occurs before the process finishes computing the result. In this case, the registration message may arrive be-

fore request termination. The case where all N processes require the future value, is the worst case scenario for lazy update strategy. In such a case, the strategy performs worse than eager message-based strategy as for each update there is an additional delay involved for the registration message to arrive. However lazy strategy can greatly reduce the number of messages exchanged. This can be a benefit in environments where network charge is an important consideration, or when future references spread but only a few nodes need the value. In counterpart, this strategy is costly in memory because future values must be stored indefinitely at the computing node.

3.5 Remarks on Semi-formal Specification of Strategies

In this chapter, we have focused on a semi-formal specification of future update strategies. We build upon the high level definitions provided in [7] to present detailed semi-formal specification of update strategies using a generalised, language independent notation. The work presented here has been published in [10]. We believe that our notation is flexible enough to be applicable to other frameworks that use first class futures as well.

To better understand the costs and trade-offs involved in each strategy, and to study their efficiency, we present a basic cost model to evaluate each of the presented strategies. Based on the specification presented here, we have implemented the update strategies and have verified the results of our cost analysis with experimental evaluation. The details of the implementation and the experiments appear in Chapter 4.

Our main contributions presented in this chapter are:

A generic event-like notation. We present a general (language independent) notation for modelling future update strategies. Our chosen notation interprets the future update strategies as combination of *events* and *operations*. Events are triggered by the middleware, whereas operations are invoked by the strategies. Although, we present three main update strategies, we believe that our notation is flexible enough to express other strategies as well that can be envisioned, including hybrid strategies obtained by mixing our presented strategies.

Semi-formal specification of three update strategies We provide detailed semi-formal specification of three main future update strategies using our generic notation. Consequently, other frameworks involving first class futures (Creol, AmbitentTalk, etc.,) can potentially benefit from our work. For example, the future update mechanism used in Creol is somewhat similar to eager message-based update strategy presented here. Our Semi-formal specification is precise-enough and contain sufficient details on underlying data-structures to be used as basis for a real implementation. We present one such implementation of future update strategies in Chapter 4. We believe that such semi-formal approaches strike a good balance between the ambiguities

inherent in informal descriptions, and the complexities of formal mathematical notations.

Cost analysis of the strategies. For better understanding of the strategies and the relative costs (in terms of number of messages and time) involved, we presented a simplified cost analysis of the protocols. We estimated the costs using indicators such as *total number of messages exchanged, time to update a given future* and *time to transmit a given result to all processes with corresponding future*. We analysed those indicators on parameters such as *number of intermediate hops/processes*, *possible number of concurrent updates*, and *the time required for serialising the payload.* The goal was to provide directions on a possible way of analysing the strategies; we consider a more detailed (and consequently more complex) cost model to be out of scope of this thesis.

The semi-formal specification presented in this chapter is used as the basis of our implementation of the strategies in ProActive. We have carried out experiments to study the efficiency of future update mechanisms and have verified the estimations presented in this chapter with experiments. Together with the implementation and experiments presented in Chapter 4, our cost-model helps in understanding which strategy is more suitable for a given application. Further discussion on suitability of strategies is presented in Section 4.4.

Chapter 4

Implementing Future Update Strategies in ProActive

Contents

4.1 Background: First Class Futures in ProActive 66

4.1.1 First Class Futures in ProActive: Automatic Continuation . . 68

4.2 Missing Future Update Strategies 70

4.2.1 Eager Message-based Strategy 70

4.2.2 Lazy Message-based Strategy 72

4.3 Experimental Evaluation 74

4.4 Concluding Remarks on Future Update Strategies 79

ProActive [15] is a distributed programming library based on ASP-calculus. Details on ASP were given in Section 2.4.3 and Section 3.1). ProActive is an implementation of ASP, and therefore contains similar notions of activities and asynchronous communication with futures. Futures in ASP and ProActive are created implicitly as placeholders for asynchronous method invocations. Futures are first class entities; futures can be safely communicated between remote processes. In this chapter, we discuss an implementation of future update mechanisms. Chapter 3 presented our semi-formal specification of future update strategies. Our semi-formal specification described in detail the working of the presented strategies and the underlying data-structures required for implementing them. Here we build on those specifications, and provide an implementation of future update protocols in ProActive.

By default, ProActive provides two operating modes related to futures: *no automatic continuation mode* (configured in ProActive descriptor) disables the use of first class futures. Alternatively, the programmer can decide to use first class futures, which are supported using *eager forward-based* strategy. Our implementation extended the support for first class futures in ProActive, and provides two additional future update strategies; *eager message-based* and *lazy message-based* strategies. Our implementation is an extension of work presented in [16], and add supports for *nested futures*, newer configuration options, along with numerous other modifications necessary to work with newer versions of ProActive. The experiments presented were carried out in collaboration with authors of [16].

ProActive is a vast library and provides a rich feature set for distributed application development. Consequently, it is not possible to cover all features of ProActive in this chapter. Here, we only focus on those aspects of ProActive which are related to futures and future update strategies. We treat everything else which is not essential for our implementation and experiments, as being out of scope of this thesis. Finally, our implementation uses ProActive version 3.9. ProActive library is being continuously updated and new feature are added regularly. However, the packages dealing with the future updates more or less have remained unchanged. Therefore, we believe that our implementation can be adapted to newer versions of ProActive without too much effort.

The chapter is organised as follows: Section 4.1 presents ProActive and shows how futures are created and manipulated in ProActive. Section 4.2 gives the implementation details for the three previously presented strategies. In Section 4.3, we present some experiments and their results to validate the working of the strategies, and to further establish the analytical conclusions presented in Section 3.4. Finally, Section 4.4 positions our work in context of previously done work and provides concluding remarks on future update strategies.

4.1 Background: First Class Futures in ProActive

ProActive is a programming library which provides a middleware for distributed application development. The library comprises notions such as concurrent active

objects, communication via asynchronous messages and first class futures. Although ProActive provides a vast array of features and constructs to support distributed programming, it is built upon a relatively small number of core notions derived from ASP-Calculus. In a way at its core, ProActive programming library may be considered an implementation of ASP-Calculus.

Like ASP-calculus, ProActive is based on the notion of *Active Objects*. An active object has its own concurrent execution thread, and is analogous to a traditional process. All communications between active objects is via asynchronous messages with futures; futures are placeholder objects for the result of the asynchronous communication. Figure 4.1 shows the anatomy of an active object in ProActive.

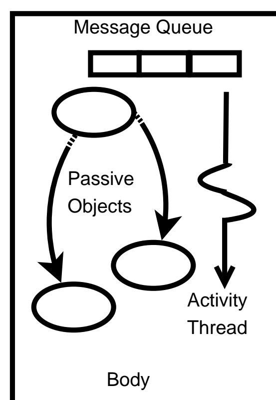

Figure 4.1: Anatomy of an Active Object

Each active object has a associated *body*, a *message queue* for storing incoming messages/requests and an execution thread – the *activity thread*. The body of an active object encodes the business logic for an active object. Additionally, the body provides support for services such as distribution, fault tolerance, migration, etc. As stated in previous chapters, all communications between active objects take the form of asynchronous method invocations–which we refer to as requests. The message queue is used for storing incoming requests. Requests are stored in the message queue until an activity thread decides to execute/serve them. The activity thread is the concurrent execution thread of an active object; it serves the requests in the message queue one by one according to some defined policy. An example request serving policy could be to serve request in FIFO order.

Figure 4.2 illustrate a ProActive program. Active objects may be created using `newActive( )` (line 1) method call. The method `newActive` is used for creating new active objects from scratch, as opposed to `turnActive( )` call, which allows for converting an existing simple object into an active object. The arguments to `newActive` describe the java class from which an active object should be constructed, and on which Java Virtual Machine (JVM) it should be deployed.

Once an active object has been created and initialised, ProActive syntax makes no distinction between active objects and simple objects. The same '.' notation is used in both cases to invoke method calls. Line 2 invokes a method `obj.foo( )` on

the newly created active object. As stated before, the communication between active objects is asynchronous with futures. In a standard java program, the execution is blocked until the completion of the method call `obj.foo( )`, which returns the value `v1` of class `V`. In ProActive, this method call is wrapped in a *request* object which is placed in the message queue (also referred as request queue) of the active object `obj`. The calling active object receives a future `v1` and can continue its execution, without waiting for the method call `obj.foo( )` to be served. It should be noted that similar to ASP_{fun} [51], here the type of the future is the same as the type of the value returned by the asynchronous invocation. This allows transparent handling of the futures.

Line 3 shows a second method call on the `obj` active object. At this stage, the message queue of `obj` contains two requests, corresponding to asynchronous requests `obj.foo` and `obj.bar`. On the invoker side, we have two future `v1` and `v2`, which are placeholders for the yet to be served requests. The execution may continue without blocking until the line 10, when `v1.bar( )` is encountered. `v1` is an unresolved future, i.e., there is no result available for this future yet. As a consequence, invoker is blocked and we refer to this as a wait-by-necessity. Again, similar to ASP-calculus and ASP_{fun}, access to an unresolved future is a blocking operation in ProActive –*wait-by-necessity* in previous chapters. The invoker remains blocked until the request is served by active object `obj` and results are available for the future `v1`.

```
A  obj = newActive(A.class, [...], null);   //line 1
V   v1    = obj.foo (param);                //line 2
V   v2    = obj.bar (param);                //line 3
. . .
. . .
v1.bar( ); // wait-by-necessity            // line 10
```

Figure 4.2: Active objects and futures in ProActive

4.1.1 First Class Futures in ProActive: Automatic Continuation

Futures in ProActive are first class objects, i.e., future references may be passed among active objects. This allows the future references to propagate among active objects (as was shown in Figure 3.1). Once a request is served (request execution terminates) and a computed future value is available, it has to be sent to all objects which may require the future value; this is achieved with a future update strategy. An overview of the three main future update strategies is presented in Section 3.2, while a more detailed semi-formal specification was presented in Section 3.3. Of the three presented strategies, ProActive supports the eager forward-based future update strategy.

ProActive documentation uses the term *automatic continuation* for referring to combination of future propagation and future update. In ProActive automatic con-

tinuation allows the use of first class futures, and continuing the execution without blocking for unresolved futures. However, performing a *strict operation* on an unresolved future results in invoker thread being being blocked.

Tracking futures The principle behind automatic continuation/eager forward-based strategy is fairly simple; future updates follow the same path as the futures (as shown previously in Figure 3.2). For the eager forward strategy to work, every active object has to keep track of destinations to which it forwards a particular future, i.e., every time a future is forwarded either as a request parameter or as a return value, the <future, active object> pair should be stored in some internal data structure. In the semi-formal specification of eager forward-based strategy in Section 3.3.2, this was captured by the *send future reference* rule *SendRef*. The *SendRef* rule stores the propagation information for a future $f^{\alpha\to\beta}$ in the *future recipient* list: $\mathcal{FR}_\beta(f^{\alpha\to\beta})$. The real implementation corresponding to the recipients list, is the `FuturePool` class. `FuturePool` keeps track of all outgoing futures; Every time a future is forwarded, the pair <future, destination> is registered as an automatic continuation. The `FutureMap` class keeps track of futures and their corresponding automatic continuations.

In a similar manner, at the destination, all incoming futures are tracked. The semi-formal specification uses the list $\mathcal{FL}$ on the receiver side for registering the newly arrived futures. ProActive implementation registers all incoming futures at the receiver's end in a static table, `FuturePool.incomingFutures`. As previously discussed, the incoming result values may also contain futures. ProActive allows partial requests and replies; the result value itself may have futures. As a result, futures may be *nested*, i.e., a value may contain other futures and values, which in turn contain futures and so on. ProActive allows arbitrary number of such future nesting. Consequently, whenever a value is received, all futures inside the value must be found and registered.

Future update As already discussed, the activity thread (execution thread) of an active object dequeues the requests from the message queue one by one and executes them. Active objects in ProActive are mono-threaded; there is only one such activity thread serving requests in each active object. At some stage, the request is served and a result value is produced for a given future. This value should be propagated to all the active objects which received the corresponding future.

While there is only one execution thread serving requests, to improve performance other threads may be used for communication specific tasks. The activity thread of a given active object need not spend time communicating future values. Consequently, to allow communication of future values in parallel to request execution, a separate thread `ActiveACQueue` housed inside `FuturePool` is used. The result value (represented by a `Reply` type object) and the list of destination active objects is wrapped in a `ACService` object, and is passed to the `ActiveACQueue` thread which invokes the necessary operations to transfer the result value. Finally,

once the value has been communicated to all active objects for which there is a registered automatic continuation, it may be garbage collected.

Configuration ProActive allows the programmer to select whether first class futures should be enabled for an application or not. Automatic continuation can be enabled or disabled using the the configuration file `ProActiveConfiguration.xml`. Setting the value for the property `proactive.future.ac` to 'false' disables first class futures ('true', enables first class futures and is the preset value).

```
<prop key="proactive.future.ac" value="false" />
```

4.2 Missing Future Update Strategies

As stated before, ProActive only supports the eager forward-based future update strategy. Our goal is to study the efficient transmission of results for first class futures. In order to do a comparison and to study the efficiency of various strategies, we implemented the missing strategies with ProActive(version 3.9). The implementation is based on the semi-formal specification presented in the previous chapter, and relies on similar kind of data structures presented there. We believe that our implementation is sufficient to establish the viability of using different future update mechanisms and provides a good basis for studying and comparing the three main future update strategies. However, it should be noted that this is only a good first version implementation and does not include a number of optimisations or garbage collection of computed results, etc.

4.2.1 Eager Message-based Strategy

Eager message-based strategy was presented in Section 3.2.3, while Section 3.3.3 gave a more detailed semi-formal specification of the strategy. To summarise, the eager message-based strategy is a centralised future update strategy, all updates are done by the active object computing the result (serving the request). The strategy is eager as all active objects which receive the future, are registered and all receive the future value when it becomes available. Registration can be performed either by the active object forwarding the future or by the active object receiving the future, leading to either a sender-based or a receiver-based registration policy. For this implementation, we have adapted a sender-based approach to future registration. We believe such an approach may simplify the task of garbage collection later on. However, garbage collection is currently not part of our implementation.

Tracking futures As stated earlier, in eager message-based strategy, the active object responsible for producing a result (referred to as *computing* object), is also responsible for communicating the results. This requires the computing object to know where the future references are, i.e, the remote references to the active objects

with a given future. This can be achieved via a registration mechanism, where every active object that receives the future reference is registered with the computing object. This registration can be performed either by the active object forwarding the future, or by the one receiving it. However, this raises another question: 'how to inform each object receiving a future where they should register'? The possible solutions vary from broadcasting the location of computing object to having a stable reference server holding all results. For our implementation, we chose to solve this issue by simply embedding the information about the computing object inside the future reference, or more precisely inside the `FutureProxy` object. In our implementation, the future proxy holds a remote reference of the computing object. This way every object which receives the future, can determine who will produce the result value. Therefore on creation, each future is initialised with the location of the computing object. Further discussion on this appears under *Remarks* later on.

Whenever a future is forwarded by an active object, either as method call parameter or as a return value, according to the semi-formal specification, this should be recorded in the future recipient list of the computing object ($\mathcal{FR}$ list). In our implementation we achieve this by implementing a new type of request in ProActive; `RequestForFuture`. `RequestForFuture` is essentially a registration message, sent by the active object forwarding the future to the computing object, registering the <future, destination> pair with the computing object. The registration request is generated automatically when a future object is being serialised. Consequently, the active object forwarding the future is the one to send the registration message, leading to sender-based registration. Similar to eager forward-based strategy, at the destination, all incoming futures are tracked. The semi-formal specification uses the list $\mathcal{FL}$ on the receiver side for registering the newly arrived futures. ProActive implementation registers all incoming futures at the receiver's end in a static table, `FuturePool.incomingFutures`. Also, as discussed in previous strategy, in case of incoming values, all nested futures are found and are also registered.

Each active object must handle an additional type of request, `RequestForFuture`. Messages of this types are used for registering future recipients and hence should not be added to the message-queue, which is used for storing normal requests. The `RequestReceiver(Impl)` class now looks at the incoming requests, and in case of a registration request, consults the list of already computed results (for a future $f^{\alpha \to \beta}$, this is given $\mathcal{FV}_\beta(f^{\alpha \to \beta})$ in the semi-formal specification). If a match is found then the result has already been computed and is returned immediately. In the implementation the computed results are stored in a static results table `FuturePool.computedValues`. In case the result is yet to be computed, the future-receiving active object(s) from the registration request is added to the list of destinations for that particular future.

Future updates At some stage, computing object produces the result for a given request. This corresponds to the *FutureComputed* event in the semi-formal specifi-

cation. As per specification, once the results are produced they are stored locally in `FuturePool.computedValues`. The result is also sent to all active objects registered in the $\mathcal{FR}_\beta$ list (future recipient list of the computing active object), or in case of our implementation, all active objects registered as destination for that particular future. In case there is more than one destination for a given future (future was forwarded more than once), updates can be performed concurrently. This can be achieved by using some group communication API. However, in our implementation, we use a simpler approach by having a pool of threads that performs the updates. The thread pool is provided with a *deep copy* of the result for transfer. The size of the thread pool may be configured using the ProActive configuration file.

Configuration The programmer may select the eager message-based strategy by setting the value for the key "proactive.future.eager" to 'true'. The key-values for the other two strategies must also be set to 'false'. Additionally, for both the eager message-based and the lazy message-based strategies, programmer must also specify a thread pool size, i.e., number of concurrent threads that may be created for future updates.

```
<prop key="proactive.future.ac" value="false" />
<prop key="proactive.future.eager" value="true" />
<prop key="proactive.future.lazy" value="false" />
<prop key="proactive.future.messagethreadpool.size" value="5" />
```

Remarks For the registration mechanism to work, each active object must know where to register outgoing futures. Our solution embeds the information about the computing object inside the future reference. Such an embedding of the information about location of computing active object, is not ideal for features such as active object migration. However, the solution proposed for the original active object migration problem, discussed in [7], apply here as well. Currently, when an active object migrates, it leaves behind a *forwarder*, which forwards any incoming messages to the active object. This forwarder can be easily modified to deal with registration messages as well. As active object/process migration is not the focus of this thesis, we do not discuss it further.

4.2.2 Lazy Message-based Strategy

Lazy message-based strategy has been presented in Section 3.2.4, while Section 3.3.4 gives a more detailed semi-formal specification of the strategy. To summarise, lazy strategy is an *on-demand* future update strategy. Future results are communicated only when active objects specifically asks for them. Other than the timing of registering for future value, the implementation for lazy message-based strategy resembles closely the eager message-based strategy. The active objects may forward futures without sending any registration messages. Registrations are performed only on access to unresolved futures which results in wait-by-necessity. This wait-by-necessity triggers the registration message.

Tracking futures All updates in lazy message-based strategy are performed by the computing object. Therefore following the same reasoning as before, future references (`FutureProxy`) contain the remote reference of the computing object. However, in contrast with eager message-based strategy, there is no need to register the destinations, when future references are forwarded. The sender forwards the future and needs not take any further action. The destination also only registers the future locally in `FuturePool.incomingFutures` (for local optimisations and processing), as specified for *SendRef* operation in Section 3.3.4.

Similar to eager message-based strategy, active objects must handle an additional type of request, `RequestForFuture`. Messages of this types are used for registering future recipients and hence are not added to the message-queue. The `RequestReceiver`(`Impl`) class looks at the incoming requests and for a registration request, consults the list of already computed results in the static results map `FuturePool.computedValues`. If a match is found then the result has already been computed and is returned immediately. In case the result is yet to be computed, the active object sending the registration message is added to the list of destinations for that particular future as required by the semi-formal specification.

Future updates At some stage, computing object produces the result for a given request. This corresponds to the *FutureComputed* event in the semi-formal specification. As per specification, once the results are produced they are stored locally in `FuturePool.computedValues`. The result is also sent to all active objects registered in the $\mathcal{FR}_\beta$ list (future recipient list of the computing active object), or in case of our implementation, all active objects registered as destination for that particular future.

The key difference between lazy message-based strategy and other eager strategies is the way wait-by-necessity is managed. For both the eager message-based strategy and eager forward-based strategy, access to an unresolved future triggers a wait-by-necessity; essentially blocking the execution thread until the result becomes available. In case of lazy message-based strategy, this wait-by-necessity also serves as trigger for requesting the future value. As a consequence of wait-by-necessity, a `RequestForFuture` is generated towards the computing active object, registering the request-sender with the computing object.

Reception of a request of type `RequestForFuture` is handled in the same way as in eager message-based strategy. Upon receiving the request, the list of already computed results is consulted ($\mathcal{FV}_\beta(f^{\alpha\to\beta})$). If a match is found then the result has already been computed and is returned immediately. In the implementation the computed results are stored in a static results table `FuturePool.computedValues`. In case the result is yet to be computed, the active object sending the registration request is added to the list of destinations for that particular future.

Configuration The programmer may select the lazy message-based strategy by setting the value for the key "proactive.future.lazy" to 'true'. Similar to eager

message-based strategy, the key-values for other strategies should be set to 'false'. Programmer must also specify a thread pool size, i.e., number of concurrent threads that may be created for future updates.

```
<prop key="proactive.future.ac" value="false" />
<prop key="proactive.future.eager" value="false" />
<prop key="proactive.future.lazy" value="true" />
<prop key="proactive.future.messagethreadpool.size" value="5" />
```

4.3 Experimental Evaluation

To validate our implementation and to better study the efficiency of future update strategies, in this section we present some results from experiments carried out using our implementation. Although, not fully optimised, we believe that our implementation provides a good starting point for comparing and evaluating the three main future update strategies. To this end, we adopted ProActive version 3.90. ProActive library is being continuously updated and new feature are added regularly. However, the packages dealing with the future updates have mostly remained unchanged. Therefore, we believe that our implementation can be adapted to newer versions of ProActive without too much effort.

We used a cluster of 11 nodes equipped with Intel(R) Xeon(TM) CPUs at 2.80GHz with 1 GB RAM running Linux kernel 2.6.9, at Università degli Studi del Sannio. The cluster nodes are connected via a Gigabit Ethernet link. Here we presents the results from two experiments, comparing the efficiency of the three main future update strategies.

As already discussed in Section 3.4, there are a number of factors that impact the performance of a given strategy. These factors include, size of future value, width of the network (in terms of number of intermediate active objects for eager forward strategy), number of threads available for concurrent updates for the message-based strategies, etc. With the two scenarios presented in the following, we study the impact of some of those factors.

Experiment: Tree configuration Our first experiment was carried out using a tree topology. We deployed a ProActive application featuring a tree topology where each node of the tree is an active object. We allow the future references to propagate to all nodes inside the tree and we compare the time required by the various strategies to transmit results to all leaf-level active objects. For the scope of the analysis, we kept the number of nodes making strict operations constant: only the leaf nodes of the tree make use of future value (important for lazy strategy). We vary the height of the trees from 1 to 7, while keeping the total number of active objects (nodes) fixed at 31.

Figure 4.3 shows a small example tree of height 2 with 7 nodes, 4 of which are leaf nodes. Although not part of the actual experiment, we use this miniature example tree to explain the behaviour of future update strategies; and consequently

the behaviour of curves showing the performance of the strategies in Figure 4.4. Only the four leaf nodes access the future value, triggering wait-by-necessity. Paths taken by the updates in the three strategies are shown.

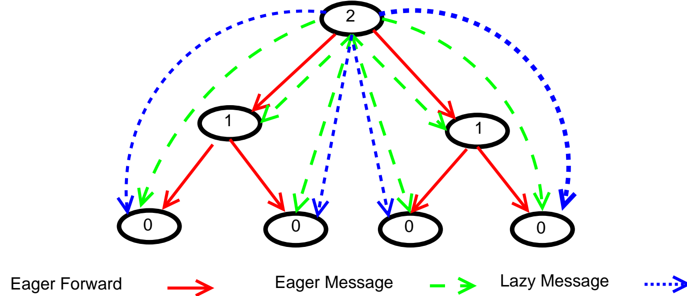

Figure 4.3: A small example tree configuration

As previously discussed, future updates in eager forward-based (shown in red) strategy must traverse along the same path as the flow of corresponding future. Consequently, in the example, future update is transmitted by the root to the nodes at height 1. Nodes at height 1 in-turn forward the value to leaf nodes (height 0). These future updates happen in parallel in the two subtrees.

The future updates for eager message-based strategy are shown in green. As per specification of eager message-based strategy, the root communicates the values to all nodes (all nodes have received the future and are registered). The updates can be done concurrently, depending on the size of thread pool.

Finally the future updates for lazy message-based strategy are shown in blue. As per specification of lazy message-based strategy, the root communicates the values to only to the leaf nodes (only leaf nodes access the future and hence are registered). Again, the updates can happen concurrently, depending on the size of thread pool.

The graph in Figure 4.4 compares the time needed to update futures for the evaluated strategies. As stated, experiments are realised over trees of varying heights. Lazy strategy (shown in blue) takes less time to update the futures since the number of updates is smaller (only leaf nodes are updated) than eager strategies (all nodes are updated). As expressed in Section 3.4 the update time for message-based strategies is independent of topology. All updates happen in centralised manner. The only factor which differentiates the performance of eager message-based and lazy message-based strategies is the number of updates to be performed. This factor is greatly impacted by the size of the thread pool. Consider Figure 4.3, with a thread pool of size 4. For a thread pool of this size, all updates for lazy strategy can happen concurrently. However, for eager-message strategy, the number of nodes to be updated is larger than the size of thread pool, requiring roughly two rounds of concurrent communications, first with 4 updates and second with 2 updates (assuming constant network latency in each case for simplifying the example). Thus eager message-based strategy takes longer than lazy message-based strategy. The

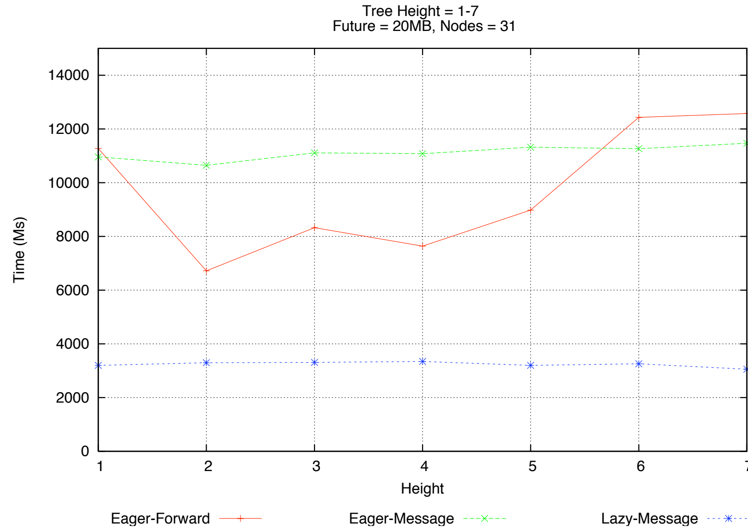

Figure 4.4: Comparison of strategies for a tree configuration

same reasoning applies to the graph in Figure 4.4. The times for different topologies remain the same for both message-based strategies, as in both cases the time for updates is independent of the tree topology.

The curve for eager forward-based strategy is more interesting. Eager forward-based strategy can take advantage of parallel updates as shown in Figure 4.3. The updates can propagate in parallel in different parts of the tree, potentially improving performance; observe the sharp drop in the red curve when moving from tree of height 1 to tree of height 2 – the updates happen in parallel in the various subtrees. However, to arrive at any leaf node, the future value must pass through a intermediate node. Each intermediate node deserialised the future value, updates the local occurrences of that future, and re-serialises the future value to transmit it to the next level. Therefore, on one hand, the strategy benefits from parallel updates in different tree segments. At the same time, the strategy suffers from having to traverse through intermediate nodes. As the height of the tree increases, more and more time is spent in serialisation-deserialisation of result value. Consequently the performance starts to degrade rapidly; this can be observed in the graph going from tree of height 4 to tree of height 7 . Therefore, eager-forward based is dependent on the topology of the tree unlike the other strategies. As the height of the tree increases, overheads increases due to time spent at intermediate nodes. Note that for tree of height 1, both of the eager strategies perform in a similar way because in that case both algorithms are roughly identical.

Experiment: Pipe configuration Our second experiment presented here, was carried out using a pipe topology. We deployed a ProActive application featuring a pipe/chain of varying length where each node of the pipe is an active object. We

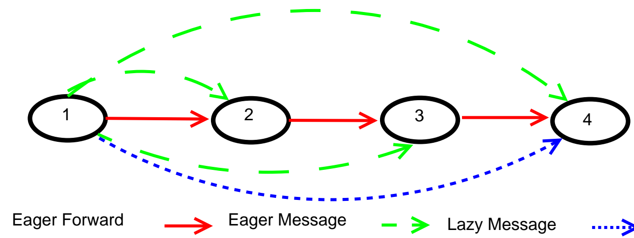

Figure 4.5: Pipe of varying length

allow the future references to propagate to all nodes along the length of the pipe, and we compare the time required by the various strategies to transmit results to the farthest(last) node of the pipe.

The purpose of this experiment is to show-case the worst case scenario for eager forward-based strategy. For the scope of the analysis, only the last element in the pipe accesses the future value (for lazy strategy). We vary the length of the pipe/object-chain from 1 to 30. In the given configuration, there is no parallelism for eager forward-based strategy. Only one update can be performed at a given time. To better compare the performance of the three strategies, we restrict the thread pool for both the message-based strategies to a single thread. Only one thread can be created to perform the updates for message-based strategies. This also allows us to compare the impact of 'number of updates to be performed' on the two eager strategies. Given that only one update can be performed at a time, this contrasts the single-step update approach of eager message-based strategy to the traverse-the-chain approach of eager forward-based strategy. Finally, for this experiment, we only consider the first future, to be computed by $node_1$ and propagated to all nodes of the pipe. Each intermediate communication forwarding the future, generates a new future; those intermediate futures are ignored while measuring the time.

Figure 4.5 shows a pipe of length 4, one of the pipes used in this experiment. We use this pipe to explain the behaviour of future update strategies; and consequently the behaviour of curves showing the performance of the strategies in Figure 4.6. Only the last node in the pipe accesses the future value, triggering a wait-by-necessity. Paths taken by the updates in the three strategies are shown.

As previously discussed, future updates in eager forward-based strategy (shown in red) must traverse along the length of the pipe to reach the last node. The future update is transmitted by the $node_1$ to $node_2$ $\ldots node_{n-1}$, finally arriving at $node_n$ (where n = length of the pipe). In Figure 4.5, the future value arrives at $node_4$ via 3 communications (hops), shown in red.

The future updates for eager message-based strategy are shown in green. As per specification of eager message-based strategy, the computing node communicates the values to all nodes (all nodes in the pipe have received the future and are registered). In Figure 4.5, for message-based strategies, $node_1$ is the computing node. The three updates for this strategy are all done by $node_1$, but one at a time (size of thread pool is 1).

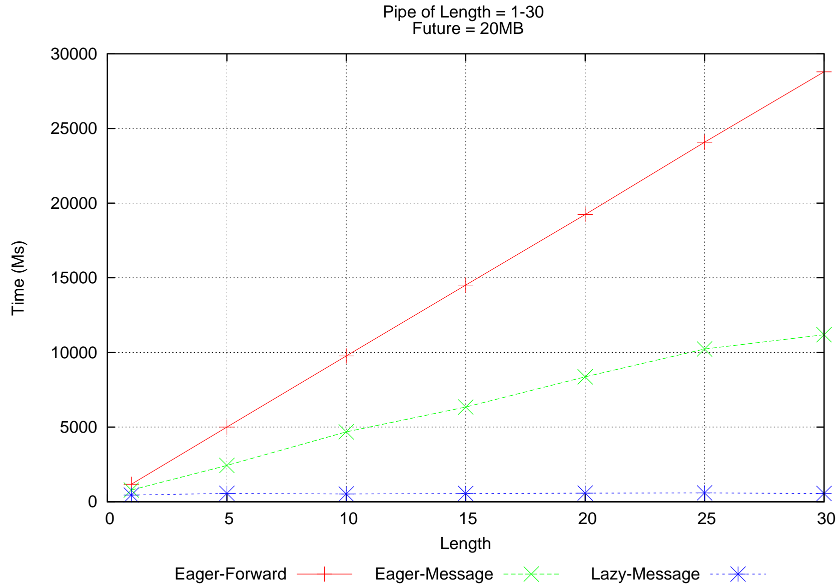

Figure 4.6: Comparison of strategies for a pipe configuration

Finally the future updates for lazy message-based strategy are shown in blue. As per specification of lazy message-based strategy, $node_1$ communicates the values only to the final node in the chain (only node to access the future and hence the only registered node). Consequently, there is only one update communication by lazy message-based strategy.

Figure 4.6 shows the time necessary to update a future along the pipe. For this experiment, we are more interested in the performance of the eager strategies. As already discussed, the size restriction on the thread pool (single thread) and the pipe topology removes any gains from concurrent or parallel transfers. Both eager forward-based and eager message-based strategies scale in a linear manner, although the eager message-based strategy scales better. Even without concurrent updates, eager message-based strategy has an advantage. There are no intermediate nodes, and results are communicated directly to each node (one at a time), as shown by green arrows in Figure 4.5.

Future updates in eager forward-based strategy go through all the intermediate nodes before arriving at the last node in the chain (shown in red). This introduces additional delays with each extra hop for forward-based strategy, with increase in the length of pipe. Finally, for lazy message-based strategy, there is only one single update required (shown in blue). As the update in done directly, in a single-step, the length of the pipe is irrelevant (the only possible affect could be the timing of future forward, and consequent access to trigger the update). As a result, the same amount of time is taken by the update for all lengths between 1 and 30, as shown by the flat horizontal line in the graph (Figure 4.6).

4.4 Concluding Remarks on Future Update Strategies

First class futures require additional support mechanisms to ensure that future results can be communicated to where they are required. We refer to such mechanisms as future update strategies. The three main future update strategies discussed in this thesis are presented in detail along with their semi-formal specification in Section 3.2 and Section 3.3. ProActive programming library, only supports the eager forward-based strategy out of the box. To better evaluate the various future update strategies, and when a particular strategy might be more suitable, we have implemented the missing strategies in ProActive. This allows us to carry out experiments comparing the efficiency of each strategy, further validating the results and analysis presented in Section 3.4.

The work presented in this chapter, is an extension of [7] and [16]. We have extended the work done in [16], adding support for nested futures, adding new configuration options, along with numerous changes to bring the implementation in-line with ProActive(3.9). The experiments presented in this section are carried in collaboration with the authors of [16].

Finally, while our implementation is not fully optimised, we believe that it provides a good starting point for studying the behaviour of various future update strategies. We have only presented some initial results to show the viability of having multiple strategies. We hope to carry out larger scale case studies to better explore the pros and cons of various strategies in the future. Even without such a case study, we believe that the work presented in this chapter and in Section 3.4 provides sufficient justification for studying future update strategies in detail.

We think that the presented analysis and the results of experiments presented in this chapter, will help answering to the non-trivial question: "Which is the best future update strategy"? There is no single *best* strategy, rather the strategy should be adopted based on the application requirements, to summarise:

- *Eager forward-based strategy* is more suitable for scenarios where the number of intermediate processes is relatively small and the future value is not too big (due to required serialisation-deserialisation at intermediate points). Also, the distributed nature of future updates results in less overloading at any specific process. However, the performance of the strategy degrades rapidly as the number of intermediate processes increases, as more and more time is spent in serialisation-deserialisation. On the other hand, the strategy is relatively less complex to implement and the computed results can easily be garbage collected.

- *Eager message-based strategy* is more adapted for process chains since it ensures that all updates are made in relatively constant time. All updates are made by the process computing the result value. Due to the centralised nature of the updates, the computing process needs the information about the processes which holds the future reference. This can be achieved using a registration mechanism where all processes that receive the future reference are registered. Once the

results become available, they are sent to all registered processes using group communication (or concurrent thread pool) mechanisms. As a consequence of its centralised nature, eager message-based strategy may require more bandwidth and resources at the process that computes the future. The computed result for a particular future may be safely garbage collected once all the registered future recipients have received the future value and no future reference (corresponding to this result) is in transit.

- *Lazy strategy* is better suited for cases where the number of processes that require future value is significantly less than the total number of processes which receive the future. The approach tries to minimise the number of future updates by delaying the registration of future recipients until there is a wait-by-necessity, i.e, the process blocks on a future. Lazy message-based closely resembles the eager message-based strategy without the mandatory registration of future recipients. All updates are done by the computing process centrally, and in one hop. Considerable savings in network load can be achieved but this has to be balanced against the additional delay inherent in the design of lazy approach. Also, as registration messages may arrive at any time, all computed results have to be stored, which requires more memory resources.

This part of the thesis focused on specification and implementation of future update strategies. The next part formally specifies a component model and a future update strategy for that model. Such a formalisation will allow us to prove the correctness of future update mechanism.

Part II

Formal Reasoning on Components: Semantics and Proofs

Chapter 5

A Framework for Reasoning on Component Composition

Contents

5.1 Background: Isabelle/HOL 85
5.1.1 Isabelle/HOL Syntax 86
5.2 An Asynchronous Component Model with Futures 89
5.2.1 Component Model Overview 90
5.2.2 Component Structure 90
5.2.3 Communication Model 92
5.2.4 Component Behaviour 93
5.2.5 Why First Class Futures in GCM ? 94
5.3 Formalisation of a Component Model in Isabelle/HOL . . . 96
5.3.1 Component Structure 97
5.3.2 Efficient Specification of Component Manipulation 98
5.3.3 Component State 103
5.3.4 Correct Component 106
5.3.5 Basic Properties on Component Structure and Manipulation 107
5.3.6 Properties on Component Correctness 109
5.4 Runtime Reconfiguration of Components 111
5.4.1 Complete Component 112
5.4.2 Reconfiguration Primitives: Unbind and Replace 113

Component models focus on program structure and improve reusability of programs. In component models, application dependencies are clearly identified by defining interfaces (or ports) and connecting them together. The structure of components can also be used at runtime to discover services or modify component structure, which allows for dynamic adaptation; these dynamic aspects are even more important in a distributed setting. Since a complete system restart is often too costly, a reconfiguration at runtime is mandatory. Dynamic replacement of a component is a sensitive operation. Reconfiguration procedures often entail state transfer, and require conditions on the communication status. A suitable component model needs a detailed representation of component organisation together with precise communication flows to enable reasoning about reconfiguration. That is why we present here a formal model[1] of components comprising both concepts.

This chapter provides support for proving properties on component models in a theorem prover. Our objective is to provide an expressive platform with a wide range of tools to help the design of component models, the creation of adaptation procedures, and the proof of generic properties on the component model. Indeed most existing frameworks focus on the correctness or the adaptation of applications; we focus on generic properties. In this context, introduction of mechanised proofs will increase confidence in the properties of the component model and its adaptation procedures. Mechanised proofs remove the possibility of human errors that are possible with the traditional pen-and-pencil proofs; in particular removing the possibilities of proving wrong statements given that the underlying logic is consistent and valid. We start from a formalisation close to the component model specification and implementation; then we use a framework allowing us to express properties in a simple and natural way. This way, we can convince the framework programmer and the application programmer of the safety of communication patterns, optimisations, and reconfiguration procedures.

We write our mechanised formalisation in Isabelle/HOL but we are convinced that our approach can be adapted to other theorem provers. The generic meta-logic of Isabelle/HOL constitutes a deductive frame for reasoning in an object logic. Isabelle/HOL also provides a set of generic constructors, like datatypes, records, and inductive definitions; those constructs support natural definitions of new objects while automatically deriving proof support for these definitions. Isabelle has automated proof strategies: a simplifier and classical reasoner, implementing powerful proof techniques. Isabelle, with the proof support tool Proofgeneral, provides an easy-to-use theorem prover environment. For a precise description of Isabelle/HOL specific syntax or predefined constructors, please refer to the Isabelle/HOL tutorial [12].

The chapter is organised as follows; Section 5.1 presents the Isabelle/HOL theorem prover along with basic constructs and syntax of HOL. Section 5.2, covers the component models giving an informal view of our formalised components and the communication model. A framework for reasoning on components in Isabelle/HOL

[1]The formalisation of GCM presented in this chapter has been published in [13].

is discussed in Section 5.3; We present the basic component structure and the infrastructure functions and lemmas that allows us to efficiently manipulate component structure and traverse component hierarchy. Some properties on components are also presented. Section 5.4.2 present formalisation of two configuration primitives and first proofs on some related lemmas. We position our work with respect to other works on reasoning frameworks and formal semantics and provide concluding remarks in Chapter 7.

5.1 Background: Isabelle/HOL

Isabelle [94] is a generic system for formalising various logics, implemented in ML [35]; Isabelle carries a syntactical similarity with ML. Isabelle/HOL [95, 12] is a specialisation, providing a generic interactive theorem proving framework, that allows implementation of formalised object logic using Higher Order Logic (HOL). The generic meta-logic of Isabelle/HOL constitutes a deductive framework for reasoning in an object logic. Isabelle/HOL also provides a set of generic constructors, like datatypes, records, and inductive definitions supporting natural definitions ; those constructs support natural definitions of new objects while automatically deriving proof support for these definitions. Semantic properties over datatypes can be expressed in a clear manner using primitive recursion which is supported by powerful proof automation using rewriting techniques. Isabelle has a simplifier and classical reasoner, to support automated proof strategies. Isabelle, with the proof support tool Proofgeneral, provides an easy-to-use theorem prover environment.

Isabelle supports a variety of logics, organised in different theory libraries. Examples of such libraries include the *Higher-Order Logic* library, *First-Order Logic* library, etc., providing the implementation of higher order logic (HOL) and first order logics respectively. Each library is a collection of `theory`(Re's). A theory in Isabelle is analogous to a module in programming languages. Each theory contains collection of types, functions, theorems and lemmas, etc. For example, the Isabelle/HOL theory `List` provides the data type list and the various assorted functions to manipulate lists. Similarly, it provides various useful lemmas on finite lists that may be used for proofs.

Higher Order Logic (HOL) is characterised by two key differences from other logics such as first order logic (FOL). First, HOL allows quantification over predicates; second, the higher order logic allows for predicates which can take one or more predicates as arguments. This makes higher order logics more expressive as compared to first and second order logics, but undecidable; we need a theorem prover to specify how to prove HOL properties. In the following we give a brief overview of various linguistic constructs used in our proofs in Isabelle/HOL.

Our goal in this section is to provide a basic overview of Isabelle/HOL, with the aim of improving the comprehension of our formalism, in particular our proofs. For a precise description of Isabelle/HOL specific syntax or predefined constructors, please refer to the Isabelle/HOL tutorial [12].

5.1.1 Isabelle/HOL Syntax

types The Higher order logic in Isabelle/HOL is encoded in theory `HOL` and has a syntax which follows that of $\lambda-$calculus and functional programming. In addition, Isabelle/HOL is a typed logic, with a type system based on ML types. HOL provides some built-in basic types like `bool` for representing truth values and `nat` for natural numbers. Ordered pairs (a_1, a_2) are also supported and have the type $\tau_1 \times \tau_2$ where each a_i is of type τ_i. The datatype `option` allows for extending an existing type for some exceptional case. Each type has associated functions some of which appear in our formalism, like functions `fst`: extract first element, and `snd`: extract second element for ordered pairs. Using an example from our formalisation, `Fid` is an ordered pair where first element is of type `nat` and the second element is of type `Name` (a user defined type). `fid` has the type `Fid` and `snd (fid)` extracts the name.

```
types Fid = nat× Name
...
fid:: Fid
...
snd (fid)
```

It should be noted that keyword `types` only defines a synonym for some existing types. Internally, all mentions of `Fid` are replaced by the elements on the right hand side of the definition (`nat × Name`). The keyword `datatype` can be used to construct new (possibly recursive) datatypes. For example, the following piece of code declares a new datatype `ItfKind` which ranges over *Client* or *Server*. We can define fields of this datatype `ItfKind`; `kind` is of datatype `itfKind`.

```
datatype ItfKind= Client | Server
...
 kind:: ItfKind
```

Isabelle/HOL ships with a number of theories which provide important datatype and assorted functions and properties. These include the the theory `List` and `Set` both of which are used extensively in our formalisation. The theory `List` provides the datatype list for finite list of elements, while `Set` provides the implementation of Set-theory for HOL. A large number of supporting functions and properties are available both lists and sets. We have seen definition of type Fid, using the same pattern we can define the type `Value`:

```
types Value = nat × (Fid list)
```

The above piece of code defines a type `Value` as a pair, the second element of which is of list of type `Fid`. `Fid list` is a type constructor for a `list` each of whose elements has the type `Fid`. The theory `List` contains a collection of functions for manipulating lists. These include, `hd` head element of list, `tl` tail of a list, `map` apply to all elements, `filter` pattern matching, etc. Similarly, we can use type `set` for

declaring sets of a given type of elements. The theory `Set` provides operations and properties on common set theory notions such as union ∪, intersection ∩, subset ⊆, etc., as well as operations on relations such as converse, transitive closure, etc. Set comprehension can be used for defining sets ; {`x. P`} is the set of all elements that satisfy the predicate `P`. Sets in Isabelle/HOL are typed, i.e, all elements of a set share the same type τ. Finite sets are also supported. Although, Isabelle/HOL allows inductive definitions for sets, not much support is available for inductive reasoning on finite sets. Lists are extensively supported and provide convenient mechanism for inductive reasoning, simplifying the proofs. As a result, we use lists extensively in our formalisation. Finally, elements of type list are easily converted into sets using the list function `set`. This is particularly convenient in our case where we need to quantify over elements in a set representation of a list. For example, for a field *val* of type `Value`, the `set` operation returns a set representation of the Fid list, allowing us to quantify over its elements.

```
types Value = nat × (Fid list)
 ...
 val:: Value
 ...
 ∀ id ∈ (set (snd val))
 ...
```

functions Functions in Isabelle are curried and the symbol ⇒ is used for giving function type. For example, `(Fid list)⇒(Fid set)` indicates a function which takes an element of type `Fid list` as input and produces an output of type `Fid set`. In general, to apply a function f on arguments a and b we write `(f a b)`. The braces here are optional and the type of the function f is $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$. Functions on datatypes in Isabelle/HOL are usually defined by recursion. Primitive recursive functions are defined using the keyword `primrec` while non-recursive functions can be defined using `constdef`. For example, we may define a trivial function `fidSet` as:

```
constdefs fidSet:: Fid list ⇒ Fid set
 fidSet FL == (set  FL)
```

The primitive recursive functions may be defined in a similar manner. An example of such a function `getName` appears in in Section 5.3.2. Without going into details of the function, `getName` takes an input of user defined type `Component` and returns an output of the user defined type `Name`. There must at most be one equation for each constructor although the order in which they appear is irrelevant. As the datatype `Component` (Section 5.3.1) can be either Primitive or Composite, we have two equations.

```
primrec getName:: Component ⇒ Name   where
        getName (Primitive ...) = ... |
        getName (Composite ...) = ...
```

records Records are quite common datatype in functional programming. Records allow us to group multiple fields into single collection. Each field of a record has a specific type, and the names of the fields are part of the record type (the order of fields is significant). The record `Result` in the following contains two fields `fid` and `fValue` of types `Fid` and `Value` respectively. The exact significance of this record and its constituent fields appears in Section 5.3.1.

```
record Result =
 fid::Fid
 fValue:: Value
```

Records provide a number of useful operations and properties. Records define field projection as functions, e.g. for a `Result res`, `(fid res)` accesses its field `fid`.

lemmas and theorems Isabelle/HOL uses the keyword `theorem` to define theorems while keyword `lemma` is used for defining lemmas. In Isabelle/HOL lemmas and theorem are used in a similar manner and can be interchanged. In our formalisation, we use `lemma` to define intermediate and supporting properties where as major results are declared as `theorem`(s). The syntax for both lemmas and theorems is as follows:

```
theorem  Name: ...(body of theorem) or lemma  Name: ...(body of lemma)
 ...
lemma trivial_lemma:  fidSet(FL) =  (set FL)
```

Each lemma and theorem has a name through which it may be applied in a proof. `trivial_lemma` shows how a lemma may be defined. The lemma can be easily proved by using the definition of function `fidSet`.

Isabelle uses two different symbols to show implication, both of which appear frequently in our Isabelle/HOL formalisation. While both $\Longrightarrow$ and $\longrightarrow$ stand of implication, they are used in different contexts. $\Longrightarrow$ is meta-level implication and is used in our code for separating assumptions from conclusions in Isabelle. On the other hand, whenever we need to express implication inside the HOL code, we use $\longrightarrow$ for implication. Lemmas with Implication inside the HOL code can be reformatted to use meta-level implication instead. In the formalisation presented in this thesis, we always reformat our lemmas to use meta-level implication for consistency. Lemmas and theorems may have more than one assumptions, separated by conjunction $\wedge$ in HOL. At the meta-level the square brackets $[\![.. \ ; \ ..]\!]$ replaces the conjunction.

```
lemma trivial_lemma2: ⟦ f ∈ fidSet(FL);  assumption2 ⟧ ⟹ f ∈ (set FL)
```

conditional expressions and optional datatype HOL supports some basic functional programming constructs for conditional expressions. Two such construct appear in the presented formalisation. The `if` construct is shown in the following piece of code. It should be noted that b must be of type `bool` while τ_1 and τ_2 should have the same type.

```
if b then τ1 else τ2  ..(* if then else construct *)
```

The optional datatype can be used for extending an existing datatype by an additional case, representing some exception. We use it to model functions that may return either some value (`Some v`) or may indicate an exceptional case like for example, the case where no match is found or a value does not exist, etc. `None`). For example, we may use a `case` with the `option` datatype as follows:

```
primrec foo:: Fid list ⇒ Name option where
              foo [] = None |
              foo FL  = snd  (hd FL)
 ...
case foo someList of
  None ⇒ ...   (* list is empty *)
  Some N ⇒ ... (* found *)
```

5.2 An Asynchronous Component Model with Futures

A component is defined as a piece of software with well-defined server and client interfaces (also called input and output ports). To increase scalability of the model, components can be designed in a hierarchical way: each component can be composed of other components. To better benefit from the component structure,we choose a component model where components are represented and can be manipulated at run-time; this allows dynamic reconfiguration and adaptability of component-based applications. Our work on formalisation is placed in the context of the GCM (Section 2.5.8); our component model is a subset of GCM model with precise semantics, and aims at proving the correctness of GCM reference implementation in ProActive/GCM. Our component model goes one step further in the autonomicity of the components: each component is a unit of deployment and of concurrency, i.e. components only interact by asynchronous requests, each component has its own threads, and components do not share memory. In this context, structured communication impose the use of futures, empty objects representing an awaited result for such asynchronous requests. To increase asynchronism, our futures are first class; meaning a future may be passed as parameter of requests or as part of return values. As a consequence, futures spread everywhere. Under reasonable hypotheses, it has been shown that the order in which results are returned has no influence on the computation [7].

The following sections define a subset of the GCM model, which we refer to as GCM-*like* model, but with a precisely defined semantics. This model incorporates hierarchical components, and asynchronous communication with futures, it uses a request-reply model.

5.2.1 Component Model Overview

Our intent is to build a mechanised model of the GCM component model but giving it a runtime semantics so that we can reason on the execution of component application and their evolution. Thus we start by describing the concepts of the GCM which are useful for understanding the work presented in this chapter. We try to distinguish clearly structural concepts that are proper to any hierarchical component model and a runtime semantics that relies on asynchronous requests and replies. Structurally, the model incorporates hierarchical components that communicate through well defined interfaces connected by bindings. Communication is based on a request-reply model, where requests are queued at the target component while the invoker receives a future. The component model has been presented in [1, 13][2] and is detailed in the following subsections.

5.2.2 Component Structure

Our model inherits most of its structure from GCM. GCM allows hierarchical composition of components. This composition allows us to implement a coarse-grained component by composition of several fine-grained components. We use the term *composite component* to refer to a component containing one or more *subcomponents*. On the other hand, *primitive components* do not contain other components, and are leaf-level components implementing business functionality. A component, primitive or composite, can be viewed as a container comprising two parts. A central *content part* that provides the functional characteristics of the component and a *membrane* providing the non-functional operations. Similarly, interfaces can be functional or non-functional. In our work and in the following description, we focus only on the functional content and interfaces.

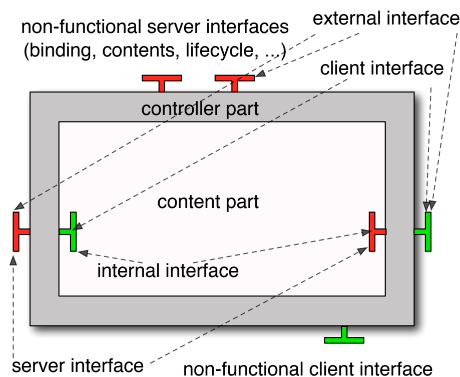

Figure 5.1: High level view of a GCM component [1]

The only way to access a component is via its interfaces. *Client* interfaces allow the component to invoke operations on other components. On the other hand,

[2]We discuss the differences between previous work[1], and our work in [13] in Chapter 7.

Server interfaces receive invocations. A *binding* connects a client interface to the server interface that will receive the messages sent by the client. For composite components, an interface exposed to a subcomponent is referred to as an *internal* interface. Similarly, an interface exposed to other components is an *external* interface. All the external interfaces of a component must have distinct names. Here, to simplify the model, each external functional interface of a composite has a corresponding internal interface. The implicit semantics is that a call received on a server external (resp. internal) interface will be transmitted – unchanged – to the corresponding internal (resp. external) client interface. Figure 5.1 shows different parts of a GCM component. The interfaces shown horizontally are the functional interfaces, while non-functional interfaces are shown vertically. Similarly, we show the server interfaces on the left, while client interfaces appear on the right. The figure also shows the various external and internal interfaces.

In Fractal[3], both the functional and non-functional external interfaces may or may not be connected to an internal interface; for example, the non-functional request may be handled by the membrane rather than sub-components in the content part. In our presented model, for simplicity, each external server functional interface has a corresponding internal interface which serves to delegate a received request to the content part of the component. Non-functional interfaces are not at all represented; we only model functional interfaces here.

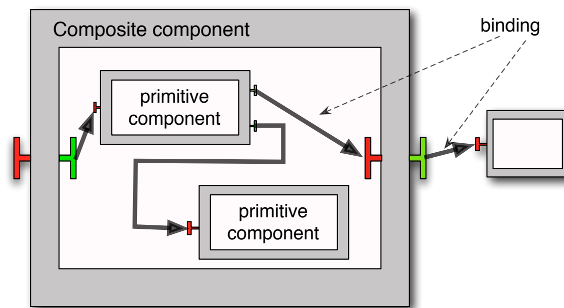

Figure 5.2: Component composition

The GCM model allows for a client interface to be bound to multiple-server interfaces; such client interfaces are referred to as multi-cast interfaces. For the moment, in our model, we restrict the binding cardinality such that bindings connect a client to a single server. Note that several bindings can anyway reach the same server interface.

Figure 5.2 shows a component system with two components, a primitive component, and a composite component along with the bindings connecting them and the various component interfaces. The composite component itself is composed of two primitive subcomponents. The requests arriving at the external server interface of the composite component are delegated to corresponding internal client interface,

[3]recall that GCM is derived from Fractal (Section 2.5.8).

which forwards the request to the server interface of the subcomponent. Similarly. the subcomponents may communicate to outside components via internal server interface of the composite.

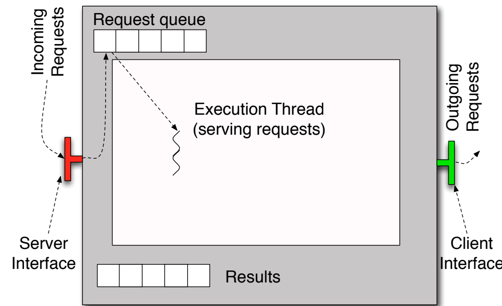

Figure 5.3: Structure of a primitive component

5.2.3 Communication Model

Our GCM-like components use a simple communication model relying on asynchronous request and replies, as presented in [1] and in Chapter 3. Communication via requests is the only means of interaction between components. We avoid shared objects or component references, and use a pass-by-copy semantics for request parameters. A component receives the requests on its external server interface. The received requests are then enqueued in the *request queue*, which holds the messages until they can be treated. Enqueuing a request is an atomic operation; implemented using rendez-vous mechanisms to ensure causal ordering of requests. Our commu-

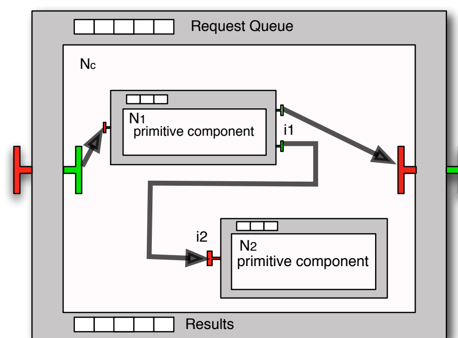

Figure 5.4: Example composite component

nication model is asynchronous in the sense that the requests are not necessarily treated immediately upon arrival. Requests are only enqueued at the target component, then the component invoking the request can continue its execution without waiting for the result. Enqueuing a request is done synchronously but the receiver is always ready to receive a request. To ensure transparent handling of asynchronous requests with results, we utilise *futures*. Futures are created automatically upon

request invocation and represent the request result, while the treatment of the request is not finished. Once the result of the computation is available, the future is replaced by the result value. Futures are first class objects: they can be transferred as part of requests or results. Figure 5.3 gives the internal structure of a primitive component. Incoming requests are enqueued in the *request queue*. The requests are dequeued and are served by one or more execution threads. when an execution thread finishes execution and a result is computed, the result is placed in the *results list* for future use. Similarly, an example of the possible structure of a composite component appears in Figure 5.4. Composite components have subcomponents and bindings in addition to the request queue and the results list. Instead of executing the requests themselves, composite components delegate the requests to the bound subcomponents for execution.

5.2.4 Component Behaviour

In our model, the primitive components represent the business logic and can have any internal behaviour. Primitive components treat all the requests they receive, choosing a processing order and the way to treat them. They can call other components by emitting a request on one of their client interface. Each primitive component must always be able to accept a request (enqueued in its request queue), and to receive a result (that will replace a future reference). Once the service of a request is finished, the produced result is stored in the computed results, which is a mapping between futures and computed values. It can then be transmitted to other components, as determined by the future update strategy [9, 14].

Figure 5.5 summarises the behaviour of a primitive component, which is governed by a labelled transition system *Behaviour*. A primitive component's behaviour allows it to: do some internal processing, emit a request towards other components, serve a request from its request-queue, transmit results towards other components, and receive results from other components. Results may only be produced for an executed request, while results from other components may arrive at any time. Formalisation of the behaviour LTS appear in Section 5.3.3; while we deal with the runtime semantics for our component model in the presence of a future update strategy in Section 6.2.1.

As opposed to the primitive components, the behaviour of a composite component is more restricted: it is strictly defined by its constituent subcomponents and the way they are composed. A composite component serves its requests in a FIFO manner, delegating them to other components bound to it. A delegated request is delivered unchanged to the target component. Overall, a request is emitted by a client interface of a primitive component, and received unchanged by the server interface of the primitive component that is (indirectly) bound to it; this request may transit through several composite components and bindings.

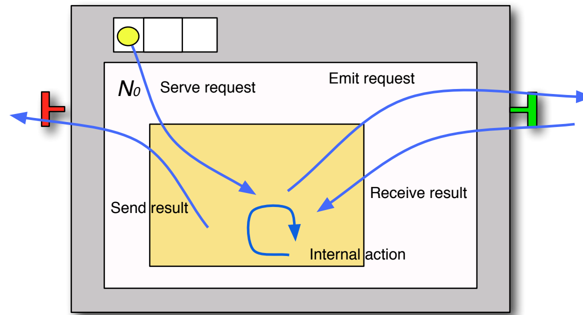

Figure 5.5: Behaviour of primitive components

5.2.5 Why First Class Futures in GCM ?

As previously stated, we formalise a subset of GCM model. It is composed of hierarchical components which communicate via asynchronous method calls. The asynchronous communication is achieved through futures. Futures are first class entities, i.e., futures may be passed as method arguments and return values; as a result futures spread everywhere. First class futures add additional complexity to the component model and its possible real world implementations. However, first class futures not only improve concurrency, but are quite necessary for mono-threaded component models like ProActive/GCM, which can be considered as one possible implementation of our formalised GCM-like model. Following example illustrates why first class futures are essential here for avoiding deadlocks.

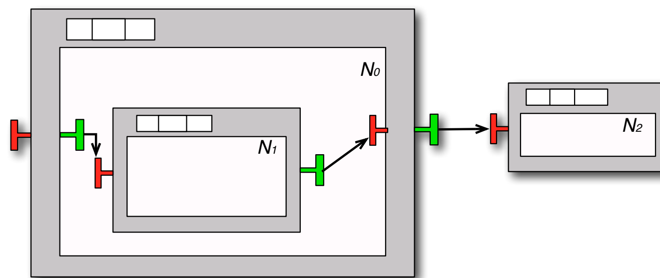

Figure 5.6: First Class Futures in GCM (a)

Consider the component system shown in Figure 5.6, consisting of two top level components N_0 and N_2. The composite component N_0 is plugged into component N_2 through its external client interface (right most green interface). The composite N_0 itself wraps a primitive subcomponent N_1. As presented in Section 5.2.2 and Section 5.2.3, the requests arriving at the external server interface of N_0 (leftmost interface shown in red), are delegated to the server interface of subcomponent N_1. Similarly, subcomponent N_1 may communicate to outside components, for example component N_2, using its external client interface (the interface shown in green on right side of N_1). Such a communication in fact, is relayed through the internal

server interface of composite component N_0 (shown in red on right side of N_0). The next couple of figures demonstrate how even this simple component system can deadlock without first class futures.

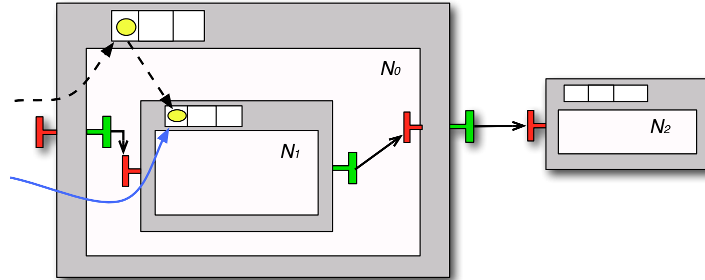

Figure 5.7: First Class Futures in GCM (b)

Figure 5.7 shows the arrival of a request (shown by a yellow sphere) at composite component N_0 intended for subcomponent N_1; the blue arrow shows the intended destination of this method call. Upon arrival at composite N_0, the request is enqueued in the message queue, while the invoker (not shown in diagram) receives a future. The execution thread of composite component N_0, dequeues the request and delegates this request to the subcomponent N_1, where it is enqueued in the message queue of N_1; N_0 receives a future as a result of this delegation. In case of futures that are not first class objects, N_0 is blocked, waiting for the actual value of the future for the delegated request.

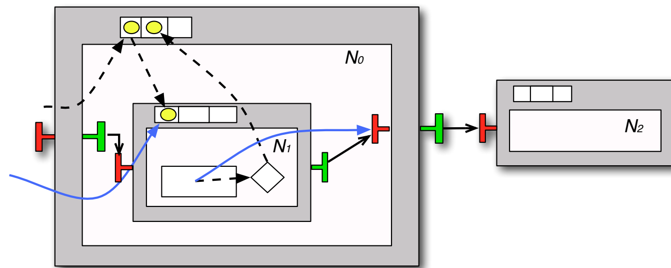

Figure 5.8: First Class Futures in GCM (c)

At some stage, the subcomponent N_1 dequeues the request and starts executing it. While executing the request, the subcomponent may attempt to communicate to external components. Such a case is shown in Figure 5.8. All communications to external components must be relayed via the composite component. The subcomponent N_1 sends a request (invokes a method) on the internal server interface of the parent composite N_0. The new request is enqueued in the message queue of N_0 while N_1 receives a future. The system is deadlocked as shown in Figure 5.9, as N_0 is still blocked waiting for the result of first future.

The subcomponent N_1 cannot finish the execution of the request unless it receives

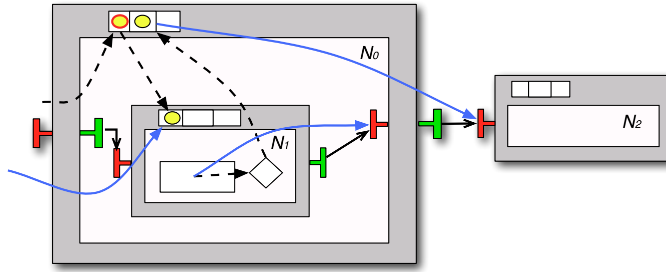

Figure 5.9: First Class Futures in GCM (d)

the reply from N_2. However, the new request cannot be delegated to N_2 because the execution thread of composite N_0 is blocked waiting for the result of the first request; resulting in a deadlock.

As opposed to mono-threaded components of ProActive/GCM, components with multiple execution threads may avoid deadlocks. However, it will still lead to a number of threads being blocked. Treating futures as first class entities, allows to side step any such situation.

5.3 Formalisation of A GCM-Like Component Model In Isabelle/HOL

We present here a framework that mechanically formalises a distributed hierarchical component model and its basic properties. We show that this framework is expressive enough to allow both the expression of component semantics and the manipulation of the component structure. Benefiting from our experiences with different possible formalisations, and from the proof of several component properties, we can now clearly justify the design choices we took and their impact[4]. The technical contributions of this chapter are the following:

- formal description in Isabelle of component structure, mapping component concepts to Isabelle constructs,
- definition of a set of basic lemmas easing the proof of component-related properties,
- additional constructs and proofs to ensure well-formedness of component structures,
- application to the design and first proofs about component reconfiguration.

We start with formalising the structure of our components. Based on the structure defined, we present some of the various *infrastructure* operations that allow us to manipulate the components for proving properties. Then we formalise additional

[4]The GCM specification framework is available at `www.inria.fr/oasis/Ludovic.Henrio/misc`

constructs to define component's state and request handling, and correctness of a component assembly. Finally we provide a set of very useful lemmas dealing with component structure and component correctness.

5.3.1 Component Structure

As we have seen in Section 5.2.2, a component in our model can either be a composite or primitive. A composite component comprises one or more subcomponents. On the other hand, a primitive component is a leaf-level component encapsulating the business logic. We define a component as:

```
datatype Component = Primitive Name Interfaces PrimState
    | Composite Name Interfaces (Component list) (Binding set) CompState
```

The above Isabelle/HOL datatype definition for `Components` has two constructors `Primitive` and `Composite`. We present below the various elements that make up the structure of our components.

Name: Each component has a unique name. We use this name as the component identifier/reference.

Interfaces: Each component has a number of public interfaces. All communication between components is via public interfaces. An interface can be either client or server and by construction a component cannot have two interfaces with the same name. Although GCM allows for both functional and non-functional interfaces, in our formalisation we focus only on functional interfaces.

Subcomponents: Composite components have a list of subcomponents, given by the `Component list` parameter. Primitive components, on the other hand are leaf-level components encoding business logic and do not have subcomponents.
Bindings: In composite components, a binding allows an interface of one component to be plugged to an interface of a second component. As shown in Figure 5.10, `(N1.i1, N2.i2)`$\in$ `bindings` if the interface `i1` of component `N1` is plugged to the interface `i2` of `N2` where `N1` or `N2` can either be component names or *This* if the plugged interface belongs to the composite component that defines the binding.
State: All components, primitive or composite have an associated state. Component state is discussed in more detail in Section 5.3.3.

Design decisions. For presenting our component model, we choose a representation that include static information like component interfaces and bindings. This allows our model to be expressive enough to support properties and proofs interleaving the component structure and more dynamic features like future update strategies. On a longer term basis it will also allow us to prove properties on component reconfiguration. In the Isabelle/HOL formalisation we chose to include the name of the component into the component itself. Like for interfaces, a first intuitive approach

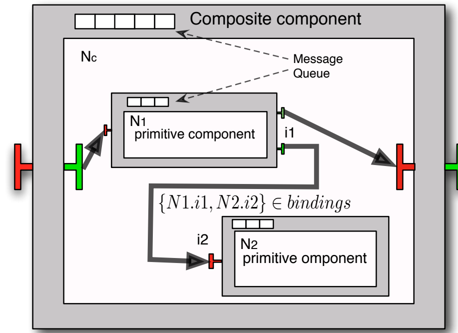

Figure 5.10: Composite Component

could be rather to define subcomponents as mappings from names to components. There are, however, major advantages to our approach. When we reason about a component we always have its name, which makes the expression of several semantic rules and lemmas more natural. The main advantage of maps is the implicit elegant encoding of the uniqueness of *Name*(s). As mentioned before, *Name*(s) are used as component references. Unfortunately, this advantage of maps is quite low in a multi-layered component model because a map can only serve one level. As we want component names to be unique globally, a condition on name uniqueness is necessary. Adding the component *Name* directly to the component makes it easier to ensure and test the uniqueness condition.

Subcomponents are defined as lists rather than finite sets because lists come with a convenient inductive reasoning easing proofs involving component structure. Of course it is easy to define an equivalence relation to identify components modulo reordering. On the contrary the bindings of a component are defined as a set because no inductive reasoning is necessary on bindings, and sets fit better to the representation of this construct.

Having a formalisation of component structure alone, although useful, is not sufficient. An adequate infrastructure needs to be developed to help in reasoning on the component model. The next section describes some of the infrastructure operations that allow us to manipulate components inside component hierarchies.

5.3.2 Efficient Specification of Component Manipulation

This section provides various operations that allow us to effectively manipulate components. These include operation for accessing component fields, mechanisms for traversing component hierarchies, and means for replacing and updating components inside the hierarchical structure. All these operations are primitive recursive functions enabling an encoding in Isabelle/HOL using the `primrec` construct. Using this construct has great advantages for the automation of the interactive reasoning process. Automated proof procedures of Isabelle/HOL, like the simplifier, are automatically adapted to the new equations such that simple cases can be solved auto-

matically. Moreover, the definitions themselves must use pattern matching leading to readable definitions.

Field access We define a number of operations for accessing various fields. These include the function *getName* and *getItfs*. The function *getName* returns the Name of the component.

```
primrec getName:: Component ⇒ Name   where
        getName (Primitive N itf s) = N |
        getName (Composite N itf sub b s) = N
```

The function *getItfs* can be used to retrieve the the component interfaces Itf. Similarly, we define getQueue, and getComputedResults for getting interfaces, request queues and replies. Requests and replies are part of the component state described in Section 5.3.3.

```
primrec getItfs:: Component ⇒ Name -~> Interfaces   where
        getItfs (Primitive N itf s) = itf |
        getItfs (Composite N itf sub b s) = itf
```

Accessing component hierarchy In order to support hierarchical components, we need a number of mechanisms to access components inside hierarchies. These range from simply finding a suitable component inside a component list to updating the relevant component with another component. The most useful of these operations are detailed below.

cpList: returns a list of all subcomponents of a component recursively. It uses the predefined Isabelle/HOL list operators # for constructing lists and @ for appending two lists. Note that the following primitive recursive function is mutually recursive and needs an auxiliary operation dealing with component lists. The final produced list is a view of the composite components and all its subcomponents recursively. The cpList of a primitive component contains one element, the primitive component itself. For a composite component, the cpList is a list constructed using the composite component itself, concatenated with the list of subcomponents of of all its subcomponents recursively. The auxiliary function cpListList, takes a component list and returns a concatenated list of CpLists of each component in the input list.

```
primrec cpList:: Component ⇒ Component list and
         cpListlist:: Component list ⇒ Component list
where
  cpList (Primitive N itfs s) = [(Primitive N itfs s)] |
  cpList (Composite N itfs subCp bindings s) =
            (Composite N itfs subCp bindings s)#(cpListlist subCp) |
  cpListlist [] = [] |
  cpListlist (C#CL) = (cpList C)@ cpListlist CL
```

cpSet: gives a set representation of the cpList of a component. This allows us to write properties in a much more intuitive way, for example, quantifying over subcomponents is easily written as ∀C' ∈ cpSet(C). Note however that a few proofs

require to stick to the `cpList` notation, while one is not sure that two identical components cannot coexist in the hierarchy. In case there are two identical components, the set representation retains only one, therefore loosing the information needed for proving uniqueness. Also, this allows us to take advantage of the various constructors for sets and make efficient use of the support for set theory in Isabelle/HOL where needed.

```
constdefs cpSet :: Component ⇒  Component set
             cpSet C == set (cpList C)
```

cpListSet: gives a set representation of a list of component lists, for all the same reasons as mentioned above for `cpSet`.

```
constdefs cpListSet :: Component list ⇒  Component set
             cpListSet CL == set (cpListList CL)
```

getCp: allows for retrieving a given component from a component list based on the component Name. The constructors `Some` and `None` represent the so-called `option` datatype enabling specifications of partial functions. Here, a component with the given name might not be defined in the list – this is nicely and efficiently modelled by a case distinction over the option type. Note the definition of ^ as an infix operator synonymous for `getCp`. This so-called pretty printing syntax of Isabelle supports natural notation of the form `CL^N = Some C'`.

```
primrec getCp:: Component list ⇒ Name ⇒ Component option (infix ‘ ^ ’)
where
   getCp [] N' = None |
   getCp (C#CL) N' = if (getName C=N') then Some C else (CL^N')
```

getSubCp: allows for retrieving a given subcomponent from a component. Note that `getSubCp` itself relies on `getCp`. The function only searches the list of subcomponents, and does not step inside the component hierarchy.

```
primrec getSubCp:: Component  ⇒ Name ⇒ Component option where
   getSubCp (Primitive N itfs s) N' = None |
   getSubCp (Composite N itfs subCp bindings s) N' = sub^N
```

getRecSubCp: allows to retrieve a subcomponent recursively from a component. Similar to `cpList`, the function `getRecSubCp` (written as `^^`) is mutually recursive and needs an auxiliary operation dealing with component lists. We again use the option datatypes to specify a partial function. The function uses the auxiliary `getRecSubCpList` to step inside each subcomponent and attempts to locate the subcomponent with the specified name throughout the component hierarchy.

```
primrec getRecSubCp:: Component  ⇒ Name ⇒ Component option (in-
fix ‘ ^^ ’) and
          getRecSubCp:: Component list ⇒ Name ⇒ Component option
where
 getRecSubCp (Primitive N itfs s) N’ = (if (N=N’) then
                                      Some (Primitive N itf s) else None)|
 getRecSubCp (Composite N itf sub b s) N’ = (if (N=N’) then
                                      Some (Composite N itf sub b s)
                                      else (getRecSubCpList sub N’) |
 getRecSubCpList [] N = None |
 getRecSubCpList (C#CL) N= (case (getRecSubCp C N) of
                          Some C’ ⇒ Some C’ |
                          None ⇒ (getRecSubCpList CL N))
```

CHANGECP: Given a component list `CL` and a component `C`, the `changeCp` function (written as `CL<-C`) replaces the component in the list `CL` having the same name as `(getName C)` by the component `C`; it does nothing if there is no component with the given name. Similarly, a `changeSubCp` function (not shown here) works with subcomponents.

```
primrec changeCp::Component list ⇒ Component ⇒ Component list (infix‘<-’)
where
  changeCp [] C = [] |
  changeCp (C#CL) C’ = if getName C=getName C’ then C’#CL else C#(CL<-C’)
```

REMOVESUBCP: Given a component `C` with Name `N`, removes the subcomponent of `C` with name `N` but does nothing if there is no subcomponent with this name. Note, here the use of a `case` switch supporting again pattern matching in Isabelle/HOL definitions.

```
primrec removeSubCp:: Component ⇒ Name ⇒ Component where
  removeSubCp (Primitive N itf s) N’ = (Primitive N itf s) |
  removeSubCp (Composite N itf sub b s) N’ = ( case sub^N’ of
            None => (Composite N itf sub b s) |
            Some C => Composite N itf (remove1 C sub) b s)
```

Manipulating requests and futures Previous operations were purely dealing with component structure. The next few operations deal with requests and futures; they allow us to express properties on the runtime behaviour of components.

COMPUTEDRQS: retrieves the list of request Ids for already computed requests in all subcomponents inside a given component, without preserving the composition order. The auxiliary function `ComputedRqsList` returns the set of request Ids for a list of components. For a primitive component, the list of computed requests can be obtained from the *computed results* attribute of component state (`PcomputedResults`). More details on component state are provided in Section 5.3.3. For a composite component, the operation is more complex and requires traversing the composition

hierarchy using the auxiliary `ComputedRqsList` function. The complete set of computed request Ids is a union of all computed requests inside the component and all its subcomponents (and their subcomponents respectively) recursively.

```
primrec ComputedRqs::  Component  ⇒   Fid set and
       ComputedRqsList:: Component list ⇒ Fid set where
 ComputedRqs (Primitive N itfs s) = (set (map fid (PcomputedResults s)))|
 ComputedRqs (Composite N itfs subCp bindings s) =
                             (set (map fid (CcomputedResults s))) ∪
                                                (ComputedRqsList subCp)|
 ComputedRqsList [] =  |
 ComputedRqsList (C#CL)=(ComputedRqs C) ∪ ComputedRqsList (CL)
```

Similar operations are needed for dealing with other aspects of requests and results. This includes operations for building lists of all referenced requests inside a component (and its subcomponents), finding a result for a given future inside a component hierarchy, etc. In all we provide almost 30 functions and predicates to help express structured component specifications efficiently.

Future registration In order to support futures, we define a number of operations for efficient manipulation of futures and for validating necessary constraints on futures. These operations are used later on in Chapter 6 for proving properties on future updates. We present here some of those operations.

The `RegisteredFuture` operation verifies if the given future is registered in a component system. More precisely, the operation checks if the component with name N is registered for future f, in the future recipient list of of some component in the component system S (`getFutrMap C` returns FRL_C).

```
constdefs RegisteredFuture:: Fid ⇒ Name ⇒ Component ⇒ bool
RegisteredFuture f N S ≡  ∃  C RL. (((S^^(snd f)) =  Some C )
                       ∧ (((getFutrMap C ) f)= Some RL) ∧ N ∈ set RL)
```

Using `RegisteredFuture`, we can now check if all the futures of a component C are registered in the component system S. `LocalReferencedRqs` gives all the futures referenced from a component, without entering its subcomponents (if component is composite).

```
constdefs LocalRegisteredFuturesComp:: Component ⇒ Component ⇒ bool
LocalRegisteredFuturesComp C S ≡ (∀ f ∈ LocalReferencedRqs C.
                                RegisteredFuture f (getName C) S)
```

Finally, we can define an operation `GlobalRegisteredFuturesComp`, to verify if all futures in a component system are registered.

```
constdefs GlobalRegisteredFuturesComp:: Component ⇒ bool
GlobalRegisteredFuturesComp  S ≡ ( ∀ C ∈ (cpSet S).
                           (LocalRegisteredFuturesComp C S))
```

Design decisions. It is crucial for the reasoning process whether one chooses lists or sets to represent various parts of the specified component structure. As we have seen above the basic infrastructure we have built up to handle our hierarchical components is mainly based on lists. Consequently, we can define operations over components and their constituents by primitive recursion and thereby decisively improve automated support. However, sets come with a more natural notation. Often set theoretic properties can be simply decided by boolean reasoning that poses no problems for logical decision procedures integrated in Isabelle/HOL, and Isabelle/HOL comes with numerous lemmas for reasoning on sets. On the other side, inductive reasoning on finite sets is less convenient than on lists. In places where we want to combine the merits of both worlds, the `cpSet`, `cpListSet`, etc., functions provides a convenient translation.

5.3.3 Component State

Our component model shall not only allow structural reasoning on hierarchical components but also reasoning about dynamic component state. While the preceding sections provided a good formalisation valid for any hierarchical component model, we now define component state in order to support communication by request and replies. Those constructs are then used to define our component semantics in Chapter 6.

Let us first focus on the high level definition of states which provide the constructs relating the component structure with the dynamic semantics[5]. We show below the two types of component states (for composite and primitive components) used in the definition of `Component` presented in Section 5.3.1.

```
record CompState =
Cqueue:: Request list
CcomputedResults:: Result list
```

```
record PrimState =
Pqueue:: Request list
PcomputedResults:: Result list
PintState:: intState
behaviour:: Behaviours
```

Each state contains a queue of pending requests, and a list of results computed by this component. Additionally, primitive components have an internal state and a behaviour for encoding the business logic, see below. We use the Isabelle/HOL `record` type constructor here; it automatically defines field projection as functions, e.g. for a `Compstate s`, `(Cqueue s)` accesses its request queue. Note that uniqueness of fields identifier required us to add a '`C`' or '`P`' prefix to fields of component states to distinguish them.

The definition of the component state relies on the definitions of requests (characterised by a future identifier, a parameter, and a target interface), and results (characterised by the future identifier and its value).

[5]The real definition of component states contains additional fields; only the more prominent fields of interest are shown here.

```
record Request =                 record Result =
id::Fid                          fid::Fid
parameter:: Value                fValue:: Value
invokedItf:: Name
```

An interesting construct is the representation of component behaviour. Each primitive component has an internal state. A behaviour specifies how a primitive component passes from an internal state to another. It is defined as a labelled transition system between internal states of a component:

The type `Behaviours` is defined as a set of triples (internal state, action, internal state). In our case actions are: internal transition `Tau`, request service `NewService`, request emission `Call`, result reception `ReceiveResult`, and end of service `EndService` which associates a result to a request. More than the precise definition of our actions, it is interesting to focus on the way behaviour can be defined and further refined by constraints. The set of possible labelled transitions can be restrained to enforce the semantics. In the piece of code below we require conditions on the internal state before and after an internal transition. We briefly look at the conditions for each of the presented actions.

TAU: the set of referenced futures can only be smaller after an internal transition, and the set of currently served requests is unchanged. The implied semantic is that `Tau` operation can only forget futures, not create new ones.

NEWSERVICE: the set of of known request ids after the transition is smaller than the set of known request ids before the transition plus any request ids (future ids) present inside the parameters. The id of the current request is added to the requests being currently served.

CALL: any new future id may be created for the result of the request, while any futures sent as part of request must be known before the transition. The set of referenced requests after the transition should be smaller than the set of referenced requests before the transition plus the new future. There is no change to the set of currently served requests.

RECEIVERESULT: the future id corresponding to the received results is removed from the list of known request ids; the ids for any new futures inside the received value however, become known after the transition. There can be no change in the currently served requests. If the received result corresponds to an unknown future, it is ignored, i.e., the state remains unchanged.

ENDSERVICE: the end of service action implies that no new futures may be created, however futures may be removed. Any futures inside the produced value must be known before the transition and results may only be produced for currently served requests. Finally, the request which has finished being served must not remain

among currently served requests.

```
typedef  Behaviours= { beh::(intState × Action × intState) set.
(∀ s s'. ((s,Tau,s')∈ beh ⟶ (set (PRqRefs s')⊆set (PRqRefs s))
                   ∧ PcurrentReqs s' = PcurrentReqs s)) ∧

(∀ s s' i v f. ((s, NewService i v f,s')∈ beh ⟶
                 (∀ v' f'. ∃ s' . (s, NewService i v' f',s')∈ beh) ∧
                 set (PRqRefs s')⊆ set (PRqRefs s)∪ set (snd v) ∧
                 (PcurrentReqs s') = f#(PcurrentReqs s) ))∧

(∀ s s' i v f.((s,Call i v f,s') ∈ beh ⟶ (
                   (∀ f'. ∃ s'' . (s,Call i v f',s'') ∈ beh)
                   ∧ set (snd v)⊆ set (PRqRefs s)
                   ∧ set (PRqRefs s')⊆ (set (PRqRefs s)∪ f)
                  ∧ (PcurrentReqs s') = (PcurrentReqs s) ))) ∧

(∀ s s' v f.((s,ReceiveResult f v,s')∈ beh ⟶ (
                   set (PRqRefs s')⊆ (set (PRqRefs s)-f)∪ set(snd v)
                   ∧ (PcurrentReqs s') = (PcurrentReqs s)
                   ∧ (f∉ set(PRqRefs s) ⟶ s=s') ))) ∧

(∀ s s' v f.((s,EndService f v,s') ∈ beh ⟶ (
                   (set(PRqRefs s')⊆ set(PRqRefs s))
                   ∧ (set(snd v) ⊆ set(PRqRefs s))
                   ∧ f ∈ set (PcurrentReqs s)
                   ∧ (PcurrentReqs s') = remove1 f (PcurrentReqs s))))
..}
```

In addition to the above mentioned constraints, we have an additional constraint requiring the component to be always able to receive results.

```
(∀ s ∈ reachable beh .∀ f. (∀ v. ∃ s'. (s,ReceiveResult f v,s')∈ beh))
```

Design decisions. Isabelle/HOL extensible records are the natural choice for representing states, requests, and results. They are better suited than simple products because they support qualified names implicitly. We did, however, not use the additional extension property of records which is similar to inheritance known from object-orientation. It could have been used to factor out the shared parts of primitive and composite components but this is not worthwhile – properties specific to the shared parts are few. Hence, there is practically no overhead caused by duplicating basic lemmas. The use of lists for requests and results is important for the efficient specification and proof of structural properties (see the design decisions in the previous section). The definition of behaviours in the internal state of primitive components uses an Isabelle/HOL type definition. This way, we can encapsulate the predicate defining the set of all well-formed behaviours into a new HOL type. These constraints are thereby implicitly carried over and can be re-invoked by using the internal isomorphism with the set `Behaviours`.

5.3.4 Correct Component

We presented the structure of our components in Section 5.2.2, while the various constructs designed to manipulate hierarchical components appear in Section 5.3.2. However, we only reason on a subset of all possible components that can be constructed according to the described component structure. We refer to this subset of components as *correct components*. Correct components are not only well-formed, but they adhere to some additional constraints. The various well-formedness rules along with the correctness constraints are presented in the following.

We start with specifying the structure of a well-formed component. A composite component is considered as correctly structured if it passes the criteria specified by the function `CorrectComponentStructure` given below.

```
primrec CorrectComponentStructure :: Component ⇒ bool where
 CorrectComponentStructure (Composite N itfs sub b s) =
   ((∀b∈bindings.(GetQualified(src b) (Composite N itfs sub b s =
                          Some (| kind=Client,cardinality=Single|))
    ∧ (GetQualified(dest b)(Composite N itfs sub b s) =
                          Some(| kind=Server,cardinality=Single|)))
    ∧ NoDuplicateSrc b
    ∧ distinct (map getName sub)
    ∧ (∀ Q∈ set (Cqueue s). (invokedItf Q)∈  dom itfs
                      ∧ kind (the (itfs (invokedItf Q))) = Server)|
...
```

A composite component has a correct structure if: each binding only connects an existing client interface to another existing server interface; each client interface is connected only once – (`NoDuplicateSrc b`); all subcomponents have distinct names – `distinct (map getName sub)` ; and all requests in the request queue of the composite refer to existing server interfaces. A primitive component has a correct structure if it follows the last requirement plus a couple of constraints relating its behaviour with its interfaces.

```
primrec CorrectComponentStructure :: Component ⇒ bool where
...
CorrectComponentStructure (Primitive N itfs s) =
             ((∀ Q ∈ set (Pqueue s). (invokedItf Q)∈ dom itfs
             ∧ kind (the (itfs (invokedItf Q))) = Server)
             ∧ (∀ s1 N v f s2. ((s1,Call N v f,s2) ∈
                          Behaviour s ⟶  N ∈ dom itfs
             ∧ kind (the (itfs N)) = Client))
             ∧ (∀ s1 N v f s2. ((s1,NewService N v f,s2) ∈
                            Behaviour s ⟶  N ∈ dom itfs
             ∧ kind (the (itfs N)) = Server)))|
...
```

And finally, we have the `CorrectComponentStructureList` constraints for component lists. A list of components is well-formed if each component inside the list has a correct structure.

```
primrec CorrectComponentStructureList :: Component ⇒ bool where
...
 CorrectComponentStructureList [] = True |
 CorrectComponentStructureList (C#CL) = (CorrectComponentStructure C
                                  ∧ CorrectComponentStructureList CL)
```

A correct component is a correctly structured component that also has uniquely defined request identifiers (`RqIdList c` gives all requests computed by `c` and its subcomponents), and all future referenced by the components should correspond to an existing request. Finally, names of all components in the composition should be unique. This differs from the well-formedness requirement which only requires the names of all direct subcomponents to be unique.

```
constdefs CorrectComponent :: Component ⇒ bool
CorrectComponent c==CorrectComponentStructure c ∧ distinct(RqIdList c)
       ∧ (ReferencedRqs c) ⊆ (set(RqIdList c))
       ∧ distinct (map getName (cpList c))
       ∧ (∀ f∈ set (RqIdList c). snd f ∈ set(map getName(cpList c)))
```

The requirement of checking correct future referencing throughout the composition hierarchy is stronger than what is needed for most proofs, and can at times be relaxed resulting in a weaker correctness requirement `CorrectComponentWeak`. `CorrectComponentWeakList` gives similar constraints but for a list of components. Using `CorrectComponentWeak` eases proofs involving component hierarchy because if a component verifies `CorrectComponentWeak` then all its subcomponents also verify it; which is not the case for `CorrectComponent`.

```
constdefs CorrectComponentWeak:: Component ⇒ bool
CorrectComponentWeak c == CorrectComponentStructure c
    ∧ distinct (RqIdList c) ∧ distinct (map getName(cpList c))

constdefs CorrectComponentWeakList:: Component list ⇒ bool
CorrectComponentWeakList CL == (CorrectComponentStructureList CL)
  ∧ distinct (RqIdListList CL)∧ distinct (map getName (cpListlist CL))
```

5.3.5 Basic Properties on Component Structure and Manipulation

In this section, we present a few properties that we proved. They deal with the constructs presented in Section 5.3.2, and are unrelated to our definition of states presented in the last section. Those lemmas are the basic building blocks on which most of our proofs rely. Out of the set of more than 80 lemmas dealing with `cpSets` and `cpLists`, we focus on the most useful and significant ones. In particular, we choose to show lemmas dealing with the `cpSet` construct because it is a higher-level one and thus reasoning on sets of components is often preferable, when possible. Note however that most of the proofs dealing with distinctness of component names will rather use `cpLists`.

We start by an easy lemma quite heavily used and very easy to prove. It states that `C` is always in `cpSet(C)` (it is proved by cases on `C`).

```
lemma cpSetFirst: C ∈ cpSet C
```

Similarly, for `cpList`, we have a corresponding lemma (again this can be easily proved by cases on `C`).

```
lemma cpListFirst : (cpList C)!0=C
```

The set of components inside a composite one can be decomposed as follows. It can be separated into the composite itself plus all the components in the `cpSet` of each sub-component.

```
lemma cpSetcomposite_set:
cpSet (Composite N itfs sub b s)={Composite N itfs sub b s}
                                 ∪ {C.∃ C'∈set sub. C∈ cpSet C'}
```

This lemma is proved by an induction on lists of subcomponents. Conversely, we can prove that, if a component is in the `cpSet` of a subcomponent of a composite, it is in the `cpSet` of the composite. We also present a more general variant of this lemma stating that if `C''` is inside `C'` and `C'` is inside `C` then `C''` is inside `C`.

```
lemma cpSetcomposite_rev:
⟦ C∈ set sub; C'∈ cpSet C ⟧⟹ C'∈ cpSet (Composite N itfs sub b s)

lemma cpSetcpSet: ⟦C''∈ cpSet C';C'∈cpSet C⟧⟹ C''∈ cpSet C
```

Although those two lemmas are very easy to prove (by induction on the component structure), they are massively used in the other proofs.

Another theorem almost automatically proved by Isabelle, but exceedingly useful is the following one. It gives another formulation of the `getCp` construct (Recall that `^` is shorthand for `getCp`).

```
lemma getCp_inlist: CL^N=Some C ⟹ C∈ set CL ∧ getName C=N
```

It is used to relate hypotheses in which a component name occurs and the component name, or the component structure. The reverse direction holds only if the component names inside `CL` are distinct as shown by the next lemma.

```
lemma getCpIdistinct:
⟦ distinct (map getName CL); getName C=N; C∈ set CL⟧⟹ CL^N=Some C
```

As the tools provided for the `distinct` construct in the Isabelle/HOL framework are a little weaker than for manipulating sets and lists, this proof is slightly longer and less automatic but still quite simple. The next lemma relates the `changeCp` primitive with the `getCp` one for the case that the name of the accessed component and the name of the changed one are different.

```
lemma upd_getCpunchanged: N≠getName C'⟹(CL <- C')^N = CL^N
```

The next lemma shows that given a component `C` and a component list `CL`, `changeCp` (written as (`CL <- C`)) only replaces the component C in the list, leaving other components unchanged. More precisely, if a component C' is inside a component list, and different component C is replaced inside that list, then C' is inside the resulting list.

```
lemma upd_diff: ⟦ C' ∈ set CL ;  getName C ≠ getName C'⟧
                                          ⟹ C' ∈ set (CL<-C)
```

The following lemma shows the usage of `CorrectComponentWeakList`, `cpListset` and `getRecSubCpList` presented in Section 5.3.2.

```
lemma getRecSubCpList_getName:
 ⟦ CorrectComponentWeakList CL; C'∈ cpListset CL⟧
                              ⟹ getRecSubCpList CL (getName C') = Some C'
```

Impact of design choices As a consequence of the mapping between component structure and Isabelle's structural support, it has been relatively easy to prove properties of component structure by automatic steps plus induction on the component structure. Consequently, the basic proofs on component sets and lists were relatively easy to handle: approximately 700 lines of code for the 80 lemmas dealing with component sets, component lists, and request identifiers, including the `getCp`, `getRecSubCp`, and `changeSubCp` primitives. By contrast, the proofs dealing with the semantics or correctness are generally much longer (several hundreds of lines per proof). However, the structural lemmas presented above are heavily used in the other proofs and strongly facilitate them.

5.3.6 Properties on Component Correctness

Based on the infrastructure for structural reasoning on the composition structure of components, we can now prove properties on the correctness of component structure presented in Section 5.3.4. The properties logically relate the degree of correctness of the structure. We present some of these lemmas here.

We start with lemmas on `CorrectComponentStructure`. The first lemma establishes the well-formedness of the subcomponents of a well-formed composite component. The reverse case `corrCompStructListComp_rev` establishes the well-formedness of all constituent components.

```
lemmacorrCompStructListComp:
 CorrectComponentStructure (Composite N  itfs subCp bindings s) ⟹
                          CorrectComponentStructureList subCp

lemma corrCompStructListComp_rev:
     ⟦ CorrectComponentStructureList subCp; C ∈ set subCp ⟧
             ⟹ CorrectComponentStructure C)
```

CorrectCompWeak establishes the relationship between CorrectComponent and CorrectComponentWeak. As stated before, it is much easier to use the weak correctness in proofs when it is sufficient.

```
lemma CorrectCompWeak: CorrectComponent C ⟹ CorrectComponentWeak C
```

The next lemma simply reiterates the uniqueness of components. A weakly correct component is unique in its cpList. Similar lemmas exists for lists of weakly correct components.

```
lemma:  correctCompWeak_distinct_name:
  CorrectComponentWeak C ⟹ distinct (map getName (cpList C))

lemma correctCompWeakList_distinct_name:
  CorrectComponentWeakList CL ⟹ distinct (map getName (cpListlist CL))
```

CorrectComponentListComp establishes the correctness of the list of subcomponents given that the parent composite component is correct. Similarly, a member of a weakly correct component list is also weakly correct.

```
lemma CorrectComponentListComp:
     CorrectComponentWeak (Composite N  itfs subCp bindings s)
                                    ⟹ CorrectComponentWeakList subCp
lemma CorrectComponentListComp_rev:
     ⟦CorrectComponentWeakList CL; C ∈ set CL⟧ ⟹ CorrectComponentWeak C
```

As a consequence, and as mentioned in Section 5.3.4, weak correctness entails weak correctness of subcomponents. Those lemmas imply that, when proving properties by induction, relying on weak correctness is very convenient as weak correctness can be used as the hypothesis of the recurrence hypothesis, which is not the case for CorrectComponent.

```
lemma SubComponent_CorrectComponentWeak:
     ⟦C'∈cpSet C; CorrectComponentWeak C⟧ ⟹ CorrectComponentWeak C'
```

The following property expresses a condition entailed in CorrectComponentWeak. C^^N returns the first subcomponent of C having the name N. If C is a weakly correct component, then there is a single component with that name, and thus the following hold:

```
lemma getRecSubCp_getName:
⟦CorrectComponentWeak C; C'∈ cpSet C⟧ ⟹ C^^(getName C') = Some C'
```

The proof of this property depends on properties on distinct names, and on the lemmas shown in this section and the preceding one.

Impact of design choices. The proofs in Isabelle/HOL are, for the most part of the correctness lemmas, almost automatic: unfolding the definitions, the proofs are mostly solved by applying the automatic tactic `auto`. Yet, these lemmas are important because they precisely relate different correctness conditions and consequently clarify subsequent proofs. They also entail properties of *compositionality*, i.e. what are the properties of a composite with respect to its constituents.

Other properties, like `getRecSubCp_getname` are harder to prove. Their proofs rely strongly on the provided infrastructure for structured components presented earlier in this section. Feasibility and readability of the proofs at the correctness level depends decisively on this clearly structured support with lemmas. Often the amount of automated proof work can be increased by adding our basic lemmas to the simplification sets of Isabelle/HOL. For example, lemma `cpListFirst` can be added as simplification rule using the `simp` attribute.

```
lemma cpListFirst [simp]: (cpList C)!0=C
```

Any proofs involving simplification will replace occurrences of `(cpList C)!0` with `C`. Similarly, attributes `intro`, `elim`, etc., may be used to annotate rules for classical reasoning.

5.4 Runtime Reconfiguration of Components

The final part of this chapter deals with runtime reconfiguration of components. Reconfiguration represents all the transformations of the component structure or content that can be handled at runtime. We consider here mainly structural reconfiguration, which includes changes of the bindings, and of the content of a component. For example replacement of a primitive component by a new one is a form of reconfiguration that allows evolution of the business code.

Reconfiguration is not the main objective of this thesis; this section serves to illustrates the expressive power of our formal representation. We show here that our framework is detailed enough to allow reasoning on component configuration-reconfiguration. More detailed proofs and formalisation of other primitives is left as future work.

Our framework enables reasoning on reconfiguration primitives and behaviour of a reconfigured component system; in part due to including the structural information in the formalisation. We illustrate below a few encodings of reconfiguration primitives and some lemmas that can be proved in Isabelle/HOL thanks to our framework. We use the the configuration primitives from Fractal [71]. For functional aspects of components, GCM inherits it's adaptive capabilities from Fractal. Reconfiguration for GCM components has been studied in more detail in [96, 97], albeit in informal settings. These primitives are:

BIND/UNBIND: allows the manipulation of component bindings.

ADD/REMOVE: allows changing the set of subcomponents of a composite component.

START/STOP: allows starting or stopping a component. Once stopped the component may be reconfigured. Stopping a component is a non-trivial task in the presence of futures; an algorithm for stopping GCM components appear in [96].

We illustrate reconfiguration capacities of our approach by defining two reconfiguration primitives and proving some related lemmas. Our proofs on reconfiguration are by no means exhaustive. Here our intent is to simply demonstrate that our framework is sufficiently detailed for specifying and proving properties on component configuration/reconfiguration. A more extensive treatment of reconfiguration primitives in Fractal can be found in [82], as part of formalisation of Fractal component model.

5.4.1 Complete Component

We take the completeness property for our components from Fractal specifications. A composite component is complete if all interfaces of its sub-components and all its internal interfaces are bound. This can be easily defined in Isabelle by the following primitive recursive predicate.

```
primrec Complete::Component ⇒ bool  and
        CompleteList::Component list ⇒ bool where
 Complete (Primitive N itf s) = True |
 Complete (Composite N itf sub bindings s) =
   (∀ C∈set sub. allExternalItfsBound C bindings) ∧
   (allInternalItfsBound (Composite N itf sub bindings s) bindings) ∧
   (CompleteList sub)

 CompleteList [] =  True |
 CompleteList (C#CL) = (Complete C ∧ CompleteList CL)
```

A primitive component is considered to be always complete; we only check completeness constraints for composite components. `(allInternalItfsBound C b)` checks that all external interfaces of `C` are bound by bindings `b`, and `(allExternalItfsBound C b)` that all internal interfaces of `C` are bound by bindings `b`. Finally, similar to `cpListlist` in Section 5.3.2, `CompleteList` recursively checks that all subcomponents are complete. Note that `Complete` and `CompleteList` are mutually recursive.

Fractal allows the definition of *optional* interfaces; an optional interface may or may not be bound. All other non-optional interfaces must be bound before a component can be started. As there is no notion of optional interface in our model (see Table 7.1), our definition is really straightforward and requires all interfaces to be bound. For a complete component, any request emitted by a component will arrive at a destination component.

5.4.2 Reconfiguration Primitives: Unbind and Replace

As stated previously, not all reconfiguration primitives have been formalised yet. We show two such reconfiguration primitives in the following.

Unbind primitive The unbind primitive removes one of the bindings defined by a composite component. Recall that only composite components have bindings.

```
primrec unbind:: Component ⇒ Binding ⇒ Component where
  unbind (Primitive N itf s) b = (Primitive N itf s) |
  unbind (Composite N itf sub bindings s) b =
          (Composite N itf sub (bindings - {b}) s)
```

The operation `unbind` removes one of the bindings from `bindings` inside the composite component. Of course un-binding does not maintain completeness, and this can be proved in our framework.

```
lemma unbinding_incomplete:
⟦b∈bindings; CorrectComponentStructure (Composite N itf sub bindings s)⟧
       ⟹ ¬ Complete (unbind (Composite N itf sub bindings s) b)
```

This lemma is proved in only 35 lines of simple Isabelle/HOL code, thanks to the properties presented in Section 5.3.5. The proof can be sketched as follows. `CorrectComponentStructure` imposes that in bindings `src b` is connected only once, thus, in `bindings-{b}`, `src b` is not connected anymore. Now, `src b` can be either `This N` if `b` connects an internal client interface to a sub-component, or of the form `CN.N` if it connects a sub-component to another interface. In the first case, the new component does not ensure `allInternalItfsBound` anymore, and in the second case, it is `allExternalItfsBound` that is not true for the component with name `CN`; note that `CorrectComponentStructure` ensures the existence of such a component.

Component replacement Let us now introduce a reconfiguration primitive that combines several basic Fractal primitives. This `Replace` primitive automatically maintains completeness.

```
primrec Replace:: Component ⇒ Name ⇒ Component ⇒ Component where
Replace (Primitive N itf s) N1 C = (Primitive N itf s) |
Replace (Composite N itf sub binds s) N1 C = addSubCp (removeSubCp
(Composite N itf sub((λb.RenameBinding b N1(getName C))‘binds) s)N1) C
```

The primitive `Replace` relies on a number of other operations, some of which appear in previous sections. `removeSubCp` removes the subcomponent of component `C` with name `N`. `addSubCp` adds a subcomponent or replaces the subcomponent if there is already a component with the same name.

```
constdefs RenameBinding:: Binding ⇒  Name ⇒ Name ⇒ Binding
RenameBinding b oldName newName ==
(| src = RenameQName (src b) oldName newName,
                      dest = RenameQName (dest b) oldName newName|)
```

RenameBinding allows one to change the src or destination of a binding. The Replace primitive maintains completeness of a correct component as expressed in the following lemma:

```
lemma replace_complete:
 ⟦sub^(getName C') = None; sub^N' = Some oldC; getItfs oldC = getItfs C';
  Complete C'; Complete (Composite N itf sub bindings s);
  CorrectComponentStructure C';
  CorrectComponentStructure (Composite N itf sub bindings s)⟧
     ⟹ Complete (Replace (Composite N itf sub bindings s) N' C')
```

This lemma requires that all involved original components are correct and complete, that the replaced component is in the composition, while the the replacement component is not in the composition, and that those two components have the same interfaces. The proof of replace_complete takes around 50 lines and relies on other simpler lemmas. One such lemma remove_diff_Name_Comp, easily proved, is shown below. The lemma simply states that given a component with correct structure there is only single component with a given name.

```
lemma remove_diff_Name_Comp:
⟦ CorrectComponentStructure (Composite N itf sub bindings s); C' ∈ set sub;
 C ∈ set (remove1 C' sub)⟧ ⟹ getName C'≠  getName C
```

A similar lemma to replace_complete proving CorrectComponentStructure for the result of the replacement operation is also proved.

```
lemma replace_correct:
⟦ sub^(getName C') = None; sub^N' = Some oldC; getItfs oldC = getItfs C';
Complete (Composite N itf sub bindings s); Complete C';
CorrectComponentStructure (Composite N itf sub bindings s);
CorrectComponentStructure C'⟧ ⟹
CorrectComponentStructure (Replace (Composite N itf sub bindings s) N' C')
```

Of course, the replace primitive can be expressed by lower level reconfiguration operations, i.e. an unbind, remove, add, bind sequence. A lemma equivalent to the preceding one could also be proved. Such a lemma would be more general but a little more complex to express because it would need to relate the set of unbound bindings, the set of re-bound ones, and the component involved in the add-remove operations.

The formalisation of reconfiguration primitives and the proofs provided here show that our framework is detailed enough to allow reasoning on component configuration-reconfiguration. In this thesis we focus on formalisation of components with future update strategies and properties on the interplay between futures

and components. More detailed proofs and formalisation of other primitives are left as future directions. Now that we have established a good understanding of our framework for reasoning on GCM-like components, the next chapter (Chapter 6) presents the runtime semantics of components in the presence of eager message-based future update strategy.

Chapter 6

Asynchronous Components with Futures : Semantics and Proofs

Contents

6.1 An Asynchronous Component Model with Futures 119

6.2 Run time Semantics for GCM-like Components 121

6.2.1 Structure and Notations 121

6.2.2 Semantics of Component Model 126

6.3 Formalisation in Isabelle and Properties 133

6.3.1 Semantics 134

6.3.2 Properties and Proofs on Eager message-based Strategy . . . 135

Chapter 5 presented a model for distributed components communicating asynchronously using futures as placeholders for results. This GCM-like component model is derived from GCM. To summarise from previous chapter, components in our component model communicate via asynchronous requests and replies where the requests are enqueued atomically at the target component, and the invoker receives a future. Then, future references can be dispersed among components. To increase asynchronism, our futures are first class; meaning a future may be passed as parameter of requests or as part of return values. As a consequence, futures spread everywhere. Under reasonable hypotheses, it has been shown that the order in which results are returned has no influence on the computation [7]. When the result is available for a future, it has to be transmitted to all interested components, as determined by a future update strategy ; discussed in Chapter 3.

Even if the execution is insensitive to the order in which futures are returned, in a real implementation of the component platform, a strategy has to be chosen to optimally perform the communication of results. We call *future update* the operation that sends a result to replace a future reference; and *future update strategies*, the different ways of performing those operations. Future update strategies have been covered in detail in Chapter 3. Formalising future updates is of little interest concerning the language properties, but it is crucial to study the implementation of this language. In order to prove the correctness of the implementation of GCM, our work aims at specifying formally future update strategies and proving correctness or efficiency properties on futures. In this chapter, we focuses on one particular strategy called *eager message-based* future update strategy (Chapter 3). We presented the formalisation of our GCM-like component model and the various infrastructure lemmas and operations required for manipulating component hierarchies in Chapter 5. Here, we build upon Chapter 3 and Chapter 5, and present the formal semantics of our component model incorporating formalisation of one future update strategy. Although we only have proofs on one such future update strategy, we present the semantics of a second strategy in Appendix B, showing the flexibility of our approach. As shown before, our model has been mechanically formalised in Isabelle/HOL, together with the proof of properties. This approach validates the actual implementation of the future update strategy itself.

Our intent is to provide a reliable and strong basis for reasoning on futures and components. For this we prove a correctness property on the registration of futures along the reduction. As already shown, the Isabelle/HOL development corresponding to this work is already consequent and shows that: our model is adequate and precise, it can be used to reason about futures and components, and the specified future update strategy guarantees basic correctness properties. This work is not restricted to the GCM component model, for example our formalisation should also provide a model for frameworks like Creol [6].

6.1 An Asynchronous Component Model with Futures

We gave an informal overview of our GCM-like component model in Section 5.2; Section 5.2.2 describes the structure of our GCM-like components, whereas Section 5.2.3 covers informally the communication model. In order to precisely define the future update strategies, the following paragraph recalls briefly key points of our component and communication model.

Component structure and behaviour Our GCM-like component model is a subset of GCM model and consequently our components inherit most of their structure from GCM. The components are hierarchical and are either primitive or composite. Primitive components are leaf-level components encapsulating the business logic and can have any internal behaviour. The external behaviour of a primitive component is captured by a *Behaviour* LTS, presented in Section 5.2.4 and more formally in Section 5.3.3. The behaviour of composite components is quite limited; essentially, a composite component serves the requests in a FIFO order, and delegates all incoming requests to its bound components. Our communication model is a request-reply model with futures; all incoming requests are enqueued at the target component in a *request queue*. All interaction between components is by means of requests. All request parameters are passed by copy semantics. Details on component structure and behaviour appear in previous chapter. A component can receive a request at any time. The requests are enqueued in the *request queue*, one or more execution threads execute (serve) these requests. On termination, results are placed in the computed *results* list for future use. The component can send requests to other components via its client interfaces. A result can arrive at any time and is used to update the corresponding future in the request queue, current component state, and the computed results list. Our futures are first class objects: no thread is blocked when a future is transferred as part of requests or results.

Future update strategy In a real implementation, updating a future value is not a simple task. Futures may be spread over a number of components, all requiring the future value. Additionally futures can appear in computed results, message queue, and current state of each component. To update all these futures efficiently, future update schemes have to be devised. The chosen scheme must ensure that any component needing a result that has been computed receives it.

First class futures can be updated using different strategies, as seen in Chapter 3. To summarise, we classified those strategies as either *eager* or *lazy*. Strategies are called eager when all the references to a future are updated as soon as the future value is calculated. They are called lazy if futures are only updated upon need, which minimises communications but increase the time spent waiting for the future value. Two eager strategies can be envisioned. *Eager forward-based* strategy, where each component remembers only the components to which it has sent the futures, and forward them the values when they become available; flow of future updates is along the same path as the futures themselves. On the other hand, in *eager*

message-based strategy, each component is responsible for sending the future value to all components which have a reference to the future. For this, all components receiving the future must register themselves as a future recipient. Finally, the *lazy message-based* strategy is the lazy version of eager message-based strategy where the future values are transferred on-demand: accessing a future reference triggers the future update.

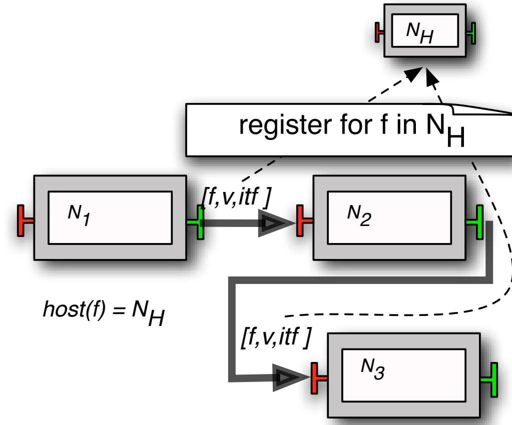

Figure 6.1: Future registration

Based on our component structure, we can derive semantics using any of the previously mentioned strategies; all three strategies are semantically equivalent, as demonstrated in ASP-calculus [7]. Eager forward-based strategy is simpler to implement as the flow of future updates follow the same path as the futures themselves. Therefore each component needs to remember only the component to which it has transferred the future. On the other hand, for the message-based strategies, the component serving the request needs to know about all components to which results should be sent. This is achieved by registering all components that require the future value with the component serving the request. Such registrations are more complex compared to simple mechanism used in eager forward strategy. Eager message-based and lazy message-based strategies are similar in nature, the only difference being the on-demand nature of component registration in lazy strategy. For eager message-based strategy, every forwarded future has to be registered with the component computing its value; including futures inside request parameters and result values. Registration mechanism for the lazy strategy is simpler because it is only triggered on future access. To conclude, eager message-based strategy is the most complex strategy and we selected it for our formalisation and proofs shown here. We do however, show the semantics of *lazy message-based* strategy in Appendix B. Our intent is to show that future update strategies can be formalised using our framework, and their properties shown. Finally, our formalisation can also be used in the context of other frameworks like Creol [6], which uses a update mechanism similar to eager message-based strategy.

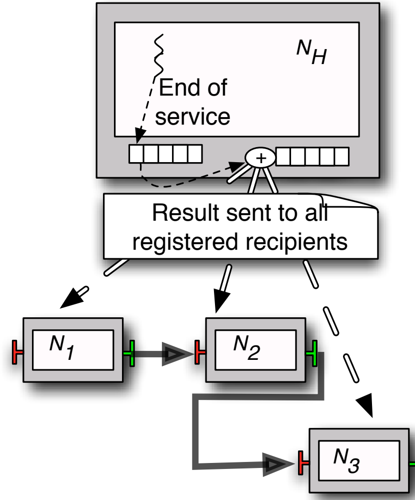

Figure 6.2: Future update

Figure 6.1 summarises the working of eager message-based future update strategy, showing the registration process for a future f which will be computed by the component N_H. Components N_1, N_2 and N_3 all have references to the future f, and consequently register with component N_H as required by eager message-based strategy. These registrations are captured in the future recipients list. Once the result is computed and placed inside the computed results list, N_H sends this result to all registered components (N_1, N_2, and N_3) as shown in Figure 6.2.

6.2 Run time Semantics for GCM-like Components

We presented our framework for reasoning on GCM-like components in Chapter 5. Here, we build on that formalisation and provide run time semantics for our components. We base our semantics on the semantics for GCM-like components presented in [1], adding precise semantics for the future update mechanism. We start by the general notations, and gradually move to more component and GCM specific notations. Similar to the reasoning framework, the component semantics are formalised[1] in Isabelle/HOL.

6.2.1 Structure and Notations

As discussed in Section 5.3.1, we make extensive use of lists and sets. We denote lists as $[a_i]^{i\in 1..n}$, while $\{a_i\}^{i\in 1..n}$ is used for a finite set. Pairs are represented with the notation (a, b). For convenience, we define a number of operators for used in our formalism. The operator # is the list append operation whereas the operator \ used as $[a_i]^{i\in 1..n} \setminus b$ removes b from the list $[a_i]^{i\in 1..n}$ whatever its position is. We use the

[1] Prototype specification available at www.inria.fr/oasis/Ludovic.Henrio/misc

notation $[a_i \mapsto b_i]^{i \in 1..n}$ to indicate a mapping from a_i to b_i. A new entry is added to an existing mapping simply by $([a_i \mapsto b_i]^{i \in 1..n})[c \mapsto d]$. $([a_i \mapsto b_i]^{i \in 1..n})[c \mapsto \emptyset\ ]$ removes the entry corresponding to c in the mapping, if it exists. It should be noted that all of the above operations have their counterparts available through various Isabelle/HOL theories.

Let f range over futures, v range over values, *itf* range over interfaces and C range over components. Additionally, we use S to denote a composite component representing the component system (all components currently instantiated). S provides context for our semantic rules; it is necessary to know the whole system to retrieve request results and to update futures and to model change in the whole component system. A future f is a pair (identifier, component name): $f ::= (id, N)$. *id* is a unique identifier for the future, while N is the name of the component computing the value of the future. Similarly, we define a value v as a pair ("object value", finite set of referenced futures): $v ::= (V, \{f_i{}^{i \in 1..n}\})$, where "object value" (V) is a structure representing the values of the underlying language but that we abstracted away by integers. This prevents values from being defined recursively; a value may contain other values. We denote (V_f, f_0) the value containing only a future reference f_0, i.e., the object consisting only of a future reference to future f_0. The second element of v, is a finite set of futures contained in the value.

Our framework allows partial results; the value v may be incomplete, i.e., may itself contain yet to be computed futures. Also, it is possible that the result value of a future f_1 is simply another future f_0, modelled by (V_f, f_0) construct. For our semantics, we chose to use a finite set instead of a list for futures inside a value; we use a `list` in Isabelle/HOL. As mentioned in previous chapter, finite sets are more suitable for our semantics and formalisation but Isabelle/HOL does not provide ample support for inductive reasoning on finite sets. Hence, we resort to lists in Isabelle/HOL. A Request R is a triple (future, value, interface): $R ::= (f, v, itf)$ where f is the future for the result of the request, v is the parameter value and *itf* is the name of target interface. In order to decouple the runtime semantics of components from Isabelle, we use these mathematical notations in place of the actual Isabelle/HOL syntax (shown in Chapter 5). The mathematical notation is not only concise but also enable us to use finite sets where needed, side stepping the implementation issues of Isabelle/HOL.

Component structure

As discussed before in Section 5.3.1, our representation of components includes static information like component interfaces and bindings. This allows our model to be expressive enough to support properties and proofs interleaving the component structure and more dynamic features like future update strategies. On a longer term basis it will also allow us to prove properties on component reconfiguration, some of which were presented in Section 5.4.2.

As seen in Section 5.2.2 and Section 5.3.1, components can be either composites

or primitives :

$$C ::= \text{Comp}[\,N,\ itfs,\ subCp,\ bindings,\ CompState\,] \mid \text{Prim}[\,N,\ itfs,\ PrimState\,]$$

All components have a unique name N (there is only one component with a given name), a list of interfaces $itfs ::= [itf_i]^{i\in 1..n}$, and a component state s. Additionally, a composite has a list of subcomponents $subCp ::= [C_i]^{i\in 1..n}$, and a set of bindings $bindings ::= \{(N_i.itf_i,\ N'_i.itf'_i)^{i\in 1..n}\}$ (See Figure 5.10). $(N_i.itf_i,\ N'_i.itf'_i)$ belongs to *bindings* if interface itf_i of component named N_i is plugged to the interface itf'_i of N'_i: where N_i and N'_i can either be a component name or *This* if the plugged interface is the composite component that defines the bindings.

Each component state s contains a request queue: $queue ::= [R_i]^{i\in 1..n}$, a list of results mapping futures to computed values: $results ::= [f_i \mapsto v_i]^{i\in 1..n}$, and a list of futures recipients: $FRL ::= [f_i \mapsto \{N_j\}^{j\in 1..n_i}]^{i\in 1..n}$. As discussed in Section 3.3, some additional constructs are required to support future update mechanism. *Future Recipient list* is one such construct and corresponds to $\mathcal{FR}$ list used in specification of future update strategies. Here, *FRL* is a list of mappings; each entry maps a future to a set of component names; these are the components which require the value for this future. We use *FRL* for tracking components which should receive the value for a given future when the value becomes available.

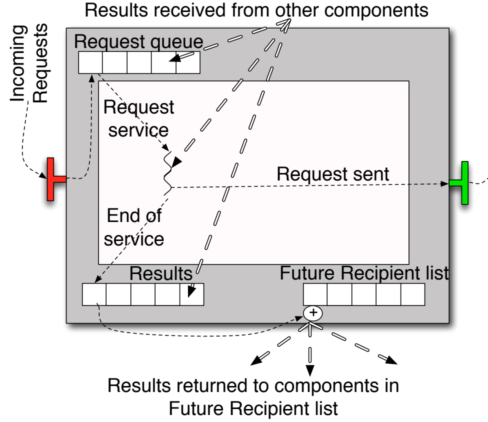

Figure 6.3: Structure and behaviour of a primitive component

Figure 6.3 captures the behaviour of a primitive component along with its structure. A primitive component state additionally contains an internal state (*intState*), and an associated behaviour *behaviour*. Behaviour, presented formally in Section 5.3.3, is a labelled transition system where the actions of the primitive components are the labels of transitions and the states are the values of *intState*. An internal state contains a list of current requests: $currReq ::= [f_i]^{i\in 1..n}$ and a list of futures referenced by the internal state: $refF ::= [f_i]^{i\in 1..n}$. Fields of a state are accessed through functions. For example, $queue(s)$ returns the current queue of the state s. Here, we use $queue(s)$ style for field access instead of the more functional style of `(queue s)` used in the Isabelle/HOL formalisation. The operation $Enqueue(C, R)$ returns the component C where its state s is replaced by

$s\,(\!|queue := queue(s)\,\#\,R\,|\!)$. In our notation, fields are modified by the operator $:=$ as shown in $s\,(\!|queue := queue(s)\,\#\,R\,|\!)$. The queue is modified to show the new queue obtained after enqueing the request R.

Additional constructs

To incorporate future update strategies in component semantics, we introduce a number of additional constructs in the formalism. We have already seen one such construct *FRL*. We now introduce a second construct, a *registration list RL*. *RL* maps a future to the set of components that require the value for this future: $RL ::= [f_j \mapsto \{N_i\}^{i\in 1..n_j}]^{j\in 1..n}$. The structure of registration list *RL* is the same as for future recipients list *FRL*. Each entry of *RL* is a mapping from a future to a list of component names; the names of components which should receive the future value when it becomes available. For eager message-based strategy to work, each component that receives a future f, must be registered with the component that will compute the value for this future ($host(f)$). We use the registration list to keep track of all such registrations and perform them during the global reduction. Details on the reduction semantics for our component model appear in Section 6.2.2. The mapping inside the registration list for a given future f is accessed by $RL(f)$. To simplify our notation, if $f \notin \mathrm{dom}(set\ RL)$ – there is no matching future in RL – then we note $RL(f) = \emptyset$. A similar semantics can be achieved in Isabelle/HOL by defining the filed access operation using the optional datatype as shown in Section 5.1.

We define three operators for manipulating lists of components. Operator ˆ is a find operation: ($subCp\hat{}N$) is the element of *subCp* which has the name N. The corresponding Isabelle/HOL function `getSubCp` has already appeared in Section 5.3.2. Similarly, operator ˆˆ is the recursive version (`getRecSubCp`), which looks for a subcomponent recursively inside the component hierarchy. Note that the two operators ˆ and ˆˆ are same as the shorthand notations defined for Isabelle/HOL functions for consistency. Operator $\leftarrow$ is the list replacement operator: ($subCp \leftarrow C_1$) replaces by C_1 the component in *subCp* that has the same name as C_1. List append operator # is overloaded for recipient list *RL* as:

$$RL\#RL' \triangleq \quad [f_j \mapsto M_j \mid f_j \in \mathrm{dom}(RL) \cup \mathrm{dom}(RL') \quad \wedge\ M_j = RL(f_j) \cup RL'(f_j)]$$

To simplify our semantics we introduce a number of support functions. Most of these have their corresponding Isabelle/HOL operations in our framework and were presented in Section 5.3.2. These functions deal with efficient manipulation of component structure.

$RqIdsSet(S)$ is the set of ids of all requests computed by S. It is the union of the domains of the request queue of S (*queue*), its currently executing requests (*currReq*), and its computed results (*results*), but also, recursively, requests computed by all its subcomponents. The corresponding Isabelle/HOL operation `ComputedRqs` is dis-

cussed in Section 5.3.2.

RefFutSet(S) is the set of all futures referenced by S and all its subcomponents recursively. It contains futures referenced in the current state (*refF*), futures in the parameters of the requests in the request queue (*queue*), futures in the the value of computed results (*results*), and futures referenced by subcomponents. The Isabelle counterpart operation is `ReferencedRqs`. By extension, we define similar function *RefFutSet*(v), giving set of futures in a value v. In Isabelle/HOL code, the operation`(snd v)` gives the same result. Finally, we define a *LocalRefFutSet*(C), which returns the set of all futures locally referenced by component C, without entering any subcomponents (if component is composite).

host(f) is the name of the component computing the value for future f, (`(snd f) in Isabelle/HOL`). $\mathit{host}(f) \triangleq \mathit{snd}(f)$.

cpSet(C) is the set formed of C and all the components recursively contained in C. *cpSet* of a primitive component is the component itself. The corresponding Isabelle/HOL operation is `cpSet`. Recall that `cpSet` is built from `cpList`.

removeResult(f, C, N) looks recursively inside the component C until a component C' with name N is found. It returns C where the state s of C' is replaced by $s \,(\!|\, \mathit{results} := \mathit{results}(s)[f \mapsto \emptyset], \mathit{FRL} := \mathit{FRL}(s)[f \mapsto \emptyset] \,|\!)$. Recall that we use component names as component identifiers.

updateFV(v, f, v') abstracts away the operation that updates a value v by replacing the occurrences of future f in value v by the new value v'. Here, as a consequence of this operation, the future f is removed from *RefFutSet*(v). The futures inside the new value v' (recall that we allow partial values) given by *RefFutSet*(v') replace the future f. A new value is returned which verifies the property:

$$\mathit{RefFutSet}\,(\mathit{updateFV}(v, f, v')) \;=\; \mathit{RefFutSet}(v) \setminus \{f\} \cup \mathit{RefFutSet}(v')$$

The corresponding Isabelle/HOL function is `UpdateFuture`, which replaces all occurrences of future f inside the value `v2` by the new value `v1`. All futures inside value `v1` are appended to remaining futures in value `v2`.

```
definition UpdateFuture:: Fid ⇒ Value ⇒ Value ⇒ Value where
 UpdateFuture f v1 v2 ≡ if (f ∈ set (snd v2)) then
                            (fst v2,(removeAll f (snd v2)) @ (snd v1))
                         else v2
```

getName(C) is the name of the component C, corresponding to the `getName` operation in Isabelle/HOL code.

$registerListFutures(S, RL)$ takes a component system S and a registration list RL and returns a new component system S' such that all the entries in the RL have been added to the recipient lists (FRL) of relevant components. More precisely, $\forall\ f.\ RL(f) \neq \emptyset$, we look inside the component system S and locate the component with name $host(f)$ inside the hierarchy. Once the component is located, we update the state s of this component such that its new state is:

$$s\,(\!| FRL := FRL(s)[f \mapsto (FRL(s)(f)) \cup RL(f)] |\!)$$

In Isabelle/HOL this corresponds to `RegisterListOfFutures` function; which locates the good component inside the component hierarchy and performs the registration for all entries in RL list (using `thelocCompAndRegFuture`). Note that due to similar reasons as before, the RL has somewhat different structure in Isabelle/HOL code (`((Fid ×Name) list)`).

```
primrec RegisterListOfFutures::Component ⇒((Fid× Name)list)⇒ Component
where
 RegisterListOfFutures C [] =  C  |
 RegisterListOfFutures C (R#L) = RegisterListOfFutures
               (locCompAndRegFuture C (snd (fst R))  (fst R) (snd R) ) L
```

Appendix A summarises the various constructs used in this chapter.

6.2.2 Semantics of Component Model

The formal semantics of our component model are given by a number of reduction relations defined by a set of inductive rules. The global reduction of the component system is $\leadsto$, it triggers either $\dashv f, v, N \mapsto_F$, or $\to_R$ reductions. $S \vdash C \to_R C'$, RL (read as C R-reduces to C', RL in context S) if in component system S, component C can be reduced to the component C'. Recall that S is a composite component that contains all instantiated components. RL contains the list of future registrations to be performed as a consequence of this reduction.

The parametrised relation $\dashv itf, f, v \mapsto_O$ (interface name, future, value) emits messages ($R ::= (f, v, itf)$). In order to be matched with a receive action, the statements $\dashv itf, f, v \mapsto_O$ are used as hypotheses to the rules for $\to_R$ for composite components. Message emission ($\dashv itf, f, v \mapsto_O$) is matched to a communication rule that enqueues the message. If $S \vdash C \dashv itf, f, v \mapsto_O C'$, then in the component system S, C emits a request on the interface itf, with parameter v, and is associated to a future f; after the emission, C becomes C'. A final parametrised relation $\dashv f, v, N \mapsto_F$ (future, value, component name) expresses that a component receives the new value v for a future f (future update message). If $C \dashv f, v, N \mapsto_F C'$, RL, then the component C with name N receives the value v for the future f. As discussed before,the value v ($v ::= (V, f_i^{i \in 1..n})$) may contain other futures; therefore N should register for all such incoming futures by adding an entry in the RL list.

In the following, we provide first the semantic rules for primitive components, followed up by the semantic rules for composite components; both semantics incorporate the formalisation of eager message-based future update strategy. Finally, we give the semantics for triggering the future update globally and the semantics for our global reduction $\rightsquigarrow$, which ensures that relevant future registrations are performed; this completes our semantics for GCM-like components with eager message-based future update strategy.

TAU

$$\frac{(\mathit{PintState}(s), \mathit{Tau}, s_2) \in \mathit{behaviour}(s)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2|\!)], [\,]}$$

CALL

$$\frac{(\mathit{PintState}(s), \mathit{Call}(i_1, v, f), s_2) \in \mathit{behaviour}(s) \qquad f \notin \mathit{RefFutSet}(S)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \dashv i_1, f, v \mapsto_O \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2|\!)]}$$

ENDSERVICE

$$\frac{(\mathit{PintState}(s), \mathit{EndService}(f, v), s_2) \in \mathit{behaviour}(s)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2, \mathit{results} := \mathit{results}(s) \# [f, v]|\!)], [\,]}$$

SERVENEXT

$$\frac{(\mathit{PintState}(s), \mathit{NewService}(v, f), s_2) \in \mathit{behaviour}(s) \qquad \mathit{queue}(s) = [f, v, i] \# Q}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2, \mathit{queue} := Q|\!)], [\,]}$$

RCVRESULTPRIM

$$\frac{\begin{array}{c}(s, \mathit{ReceiveResult}(f, v), s') \in \mathit{behaviour}(s) \\ s'' = s'(\!|\, \mathit{results} = [f_i \mapsto v_i \mid f_i \in \mathrm{dom}(\mathit{results}(s)) \wedge \\ v_1 = \mathit{updateFV}((\mathit{results}(s))(f_i), f, v)], \\ \mathit{queue} = [[f_i, v_i, \mathit{itf}_i] \mid [f_i, v_i', \mathit{itf}_i] \in \mathit{queue}(s) \wedge v_i = \mathit{updateFV}(v_i', f, v)]|\!)\end{array}}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \dashv f, v, N \mapsto_F \mathrm{Prim}[N, \mathit{itfs}, s''], [(f'', N) \mid f'' \in \mathit{RefFutSet}(v)]}$$

Figure 6.4: Primitive Component Semantics

Semantics for primitive components Figure 6.4 presents the semantic rules for primitive components. As stated previously, the external behaviour of a primitive component is given by an LTS `Behaviour`; discussed in Section 5.3.3. For each of the `Actions` in `Behaviours` there is a semantic rule which encompasses the internal action. Recall that the actions for primitive components are: internal transition, request service, request emission, result reception, and service termination.

TAU: gives semantics for a non observable internal behaviour of a primitive component. If the internal state of the component is s, and there is an internal transition *Tau* to an internal state s_2, then the internal state of the component can be changed to s_2. No further change occurs in the component. In Figure 6.3, it corresponds to the internal transitions inside the contents of the component.

Call: gives the semantics for emitting a request to another component. Given the internal state s, the call transition with the parameters i_1, v, f to the internal state s_2 requires that the future f should be a newly created future. Recall that *RefFutSet*(S) is the set of all futures referenced in S and all its subcomponents recursively. This internal transition in the behaviour results in emission of request on the interface i_1. This is captured through the parametrised reduction $\dashv i_1, f, v \mapsto_O$. The internal state is modified to s_2. This parametrised reduction will synchronise with the operation that enqueues a request on the component bound to the interface. In Figure 6.3, this corresponds to the "Request send " arrow towards right-hand side client interface.

EndService: gives the semantics for termination of request execution. More precisely, it indicates that one of the execution threads is finished executing a current request in this primitive component, producing the result value v. The pair `[f,v]` is added to the computed result list and the internal state is modified. As per requirements for the internal action *Endservice*, after the reduction, the just finished request id f is no longer in the component's list of current requests. Any futures inside the produced value v must be previously known; therefore the registration list is empty. In Figure 6.3, this rule corresponds to the arrow marked "End of Service".

ServeNext: gives the semantics for start of execution for a request. The internal action *NewService*(v, f), results in the oldest request to be added to the list of current requests in the internal state of this primitive component. f is the future associated with this request. After the reduction, the internal state of the component is modified to s_2 and the request at the head of the message queue (which is now a current request) is removed. In Figure 6.3, this corresponds to the arrow marked "Request Service".

RcvResultPrim: gives the semantics for future update for a primitive component. The behaviour of primitive components always allows for the *ReceiveResult*(f, v) transition; results may be received at any time. All references to the future f in the computed results and the message queue are replaced with the new value v. As stated before, the received result value v may contain other futures; therefore the receiving component must register for those futures as well. This is captured by making an entry in *RL* list for every future in v (*RefFutSet*(v)). Interaction between the component semantics and the internal state of the primitive is enabled by triggering internal transitions on the primitive *behaviour*, here *ReceiveResult*. The internal state of the component is changed to take into account these changes. In Figure 6.3, this corresponds to the "Results received from other components", incoming arrows.

Semantics for composite components Figure 6.7 and Figure 6.11 present the reduction rules dealing with composite components. The first rule embeds subcomponent reduction in composite contexts; the second rule allows composite compo-

nents to emit requests on their external client interfaces. The three COMM-rules define the request transmission over the three different kinds of bindings; these arise from the nature of relationship between the communicating components. Finally, the two RCV rules performs future updates inside composite components based on whether it is the right component for doing the update or not.

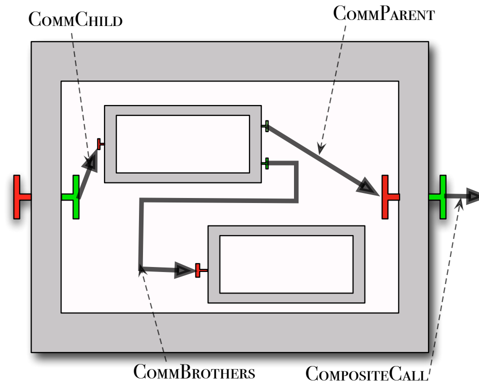

Figure 6.5: Component Communications

Figure 6.5 illustrates the different kinds of communications expressed by the COMM-rules and the composite call rule. The existence of the different form of rules is due to the component structure. Additionally, as there are three kind of bindings (from a parent component to a subcomponent, between two subcomponents, or from a subcomponent to its parent), there are three kind of communication rules (resp. COMMCHILD, COMMBROTHER, or COMMPARENT).

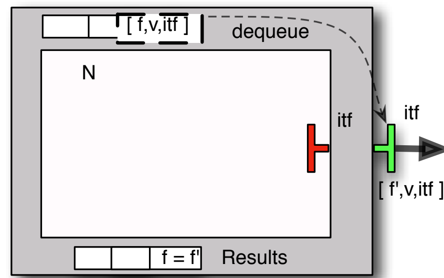

Figure 6.6: COMPOSITECALL

HIERARCHY: Hierarchy defines the compositionality of components. If a component C reduces to a component C' in isolation, then it also does so inside a composite. The registration list is the one for the subcomponent.

COMPOSITECALL: This rule describes how a composite component emits a call on the external client interface as illustrated in Figure 6.6. The request $[f, v, \mathit{itf}]$, received on internal server interface *itf*, is sent on the matching external client interface

Hierarchy
$$\frac{(subCp \uparrow N) \ = \ C \qquad S \vdash \ C \rightarrow_R \ C',\ RL}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[N_0, itfs, (subCp \leftarrow C'), bindings, s],\ \ RL}$$

CompositeCall
$$\frac{queue(s) \ = \ [f, v, itf] \ \# \ Q \qquad f' \notin RqIdSet(S) \qquad s' \ = \ s(\!| queue := Q,\ results := results(s)[f \mapsto (V_f, \{f'\})] |\!)}{S \vdash \text{Comp}[N, itfs, subCp, bindings, s] \dashv itf, f', v \mapsto_O \text{Comp}[N, itfs, subCp, bindings, s']}$$

CommBrothers
$$\frac{C = (subCp \uparrow N) \quad [N.itf,\ N'.itf'] \in bindings \quad S \vdash \ C \dashv itf, f, v \mapsto_O C' \quad host(f) = N' \quad subCp' = subCp \leftarrow C' \quad subCp'' = subCp' \leftarrow (Enqueue(subCp' \uparrow N', [f, v, itf']))}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[N_0, itfs, SubCp'', bindings, s],\ \ [(f'',\ N') \mid f'' \in RefFutSet(v)] \ \# \ [f, N]}$$

CommChild
$$\frac{queue(s) = [f, v, itf] \ \# Q \quad [This.itf, N'.itf'] \in bindings \quad f' \notin RefFutSet(S) \quad host(f') = N' \quad subCp' = subCp \leftarrow \ (Enqueue((subCp \uparrow N'),\ [f',\ v,\ itf'])) \quad s' := s(\!| queue := Q,\ results := results(s)[f \mapsto (V_f, f')] |\!)}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[N_0, itfs, subCp', bindings, s'],\ [f', N_0] \ \# \ [(f'', N') \mid f'' \ \in \ RefFutSet(v)]}$$

CommParent
$$\frac{(subCp \ \uparrow \ N) = C \quad [N.itf',\ This.itf] \in bindings \quad subCp' = subCp \leftarrow C' \quad S \vdash \ C \dashv itf', f, v \mapsto_O C' \quad host(f) = N_0}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R Enqueue(\text{Comp}[N_0, itfs, subCp', bindings, s],\ [f, v, itf]),\ \ [f, N] \ \# \ [(f'', N_0) \mid f'' \in RefFutSet(v)]}$$

RcvResultComposite(1)
$$\frac{s' = s(\!| results = [f_i \mapsto v_i \mid \exists v_i'.\, [f_i \mapsto v_i'] \in results(s) \ \wedge \ v_i = updateFV(v_i',\ f,\ v)],\ queue = [[f_j,\ v_j,\ itf_j] \mid \exists v_j'.\, [f_j,\ v_j',\ itf_j] \in queue(s) \ \wedge \ v_j = updateFV(v_j',\ f,\ v)]}{S \vdash \text{Comp}[N, itfs, subCp, bindings, s] \dashv f, v, N \mapsto_F \text{Comp}[N, itfs, subCp, bindings, s'],\ \ [(f'',\ N) \mid f'' \in RefFutSet(v)]}$$

RcvResultComposite(2)
$$\frac{N_0 \neq N' \qquad (subCp \ \uparrow \ N) \dashv f, v, N' \mapsto_F C',\ RL \qquad subCp' = subCp \leftarrow C'}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \dashv f, v, N' \mapsto_F \text{Comp}[N_0, itfs, subCp', bindings, s],\ \ RL}$$

Figure 6.7: Semantics of the component composition (a)

(with same name). This call will be matched against a Comm rule that enqueues this request. Consequently, this request will be handled by the enclosing composite. A fresh future f' is found for this new request. The composite component records that the value of f is now the new future f', and dequeues the request. The grey arrow in Figure 6.6 show the call semantics, a request received on the internal server interface is delegated unchanged to the external client interface. In Figure 6.5, this

rule corresponds to the outgoing arrow from the external client interface of the composite component.

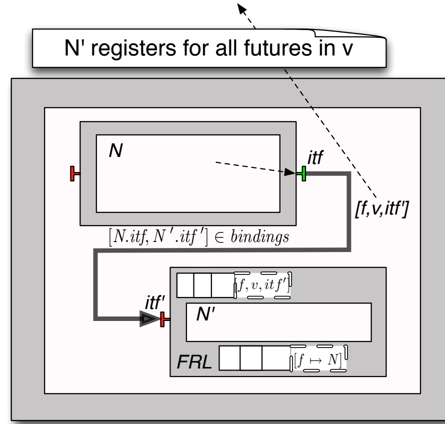

Figure 6.8: CommBrother

CommBrothers: This rule expresses communication between two sibling subcomponents of a composite component, as illustrated in Figure 6.8. If N and N' are the names of two subcomponents of component N_0, then component N can pass a call to component N' if the client interface *itf* of N is bound to the server interface itf' of N' ($[N.itf, N'.itf'] \in bindings$). The call parameters f, v are passed unchanged to interface itf' of subcomponent N'. The operation *Enqueue* is used to place the request $[f, v, itf']$ onto the request queue of the destination. N is reduced simultaneously, sending the request. Component N then registers (in the *RL* list) for receiving the result for future f when it is available. Similarly, N' also registers for all futures inside the parameter v. In Figure 6.5, this rule corresponds to the arrow (showing the binding) between the two subcomponents of the composite.

CommChild: This rule expresses request delegation between a composite component and its subcomponent as shown in Figure 6.9. The request $[f, v, itf]$ is dequeued from the request queue of the parent. A new future f' is created and added to the result list of the parent as the result for this request. The new request $[f', v, itf']$ is enqueued at the subcomponent. The exact subcomponent is determined using the bindings: a request delegated to a subcomponent necessarily arrived an external server interface, call it *itf*, if *This.itf* is bound to $N'.itf'$ then the request is sent to the interface itf' of the subcomponent N'. The component N_0 registers in the *RL* list to receive the result for f', also the destination N' registers for any future inside the request parameter v. In Figure 6.5, this rule corresponds to left most arrow binding the internal client interface of the composite with the server interface of the subcomponent.

CommParent: This rule expresses communication between a subcomponent and the composite component containing it, see Figure 6.10. When a subcomponent N

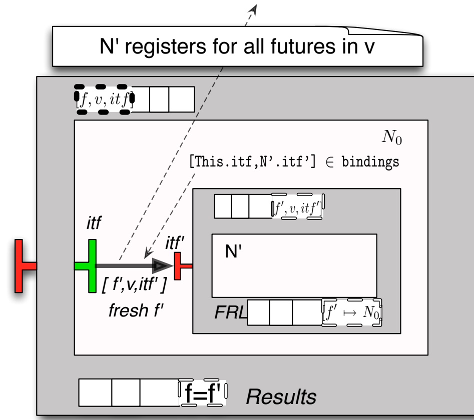

Figure 6.9: CommChild rule

of a composite component N_0 emits a request $[f, v, itf']$ to its parent, the request is added to the composite component's request queue. For this, the subcomponent interface $N.itf'$ must be bound to the parent component interface *This.itf*. The component N registers to receive the value for f when it is available; also, the values for any future inside v must be sent to N_0. In Figure 6.5, this rule corresponds to the communication between the subcomponent and the right most internal client interface of the composite component.

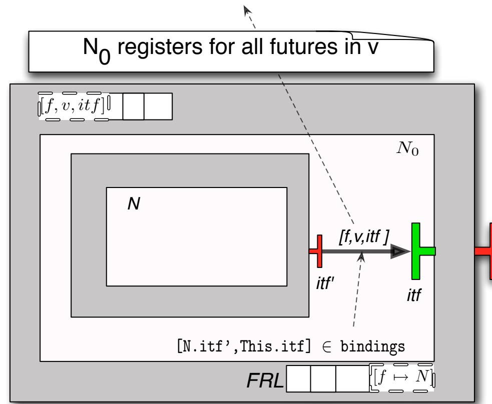

Figure 6.10: CommParent

RcvResultComposite(1): This rule expresses future update for a composite component which is the destination of the update. At the component N, the state s is updated such that the new value v for the future f, replaces the old value inside both the *results* and *queue*. The values for any futures inside v should be sent to N, this is recorded in the RL list.

RcvResultComposite(2): This rule ensures that a future update is applied at the component that is the destination of the future update, i.e., only at the component which has the same name as given in reduction parameter $\dashv f, v, N \mapsto_F$. Only the

sub-component that contains a component of name N is able to be reduced. This rule navigates in the component hierarchy, finds the component with name N, and applies rule RcvResultComposite(1)or in case of a primitive component, the rule RcvResultPrim.

Trigger future update and global reduction We have shown the semantic rules for both primitive and composite components. The final set of rules, completing the runtime semantics of our GCM-like components with first class futures appear below. Trigger future update rule defines the mechanisms for initiating future updates. The last rule Global-Reduction triggers the global reduction $\leadsto$.
TriggerFutureUpdate: This rule selects a computed result of a component C

TriggerFutureUpdate
$$\frac{C \in \mathit{cpSet}(S_o) \quad \text{the state of } C \text{ is } s \quad \mathit{results}(s)(f) = v \quad \mathit{FRL}(s)(f) = \{N_i\}^{i \in 1..n} \qquad \forall i \in 1..n,\ S_{i-1} \dashv f, v, N_i \mapsto_F S_i,\ RL_i \quad S' = \mathit{RemoveResult}(f,\ S_n,\ \mathit{getName}(C))}{\vdash S_o \leadsto \mathit{regListFutures}(S',\ RL_1 \# RL_2 \# \ldots \# RL_n)}$$

Global-reduction
$$\frac{S \vdash S \rightarrow_R S', RL}{S \leadsto \mathit{registerListFutures}(S', RL)}$$

Figure 6.11: Semantics of the component composition (b)

in the component system S_o for initiating the future update process. The value v for the future f, has to be sent to all components (${N_i}^{i \in 1..n}$) in future recipient list FRL for the future f ($FRL(f)$). For every component N_i, a future update is triggered on N ($host(f)$, this is matched by a *RcvResult* rule.
Global-reduction: From $\rightarrow_R$, a reduction $\leadsto$ for the global component system can be performed, after performing all future registrations.

6.3 Formalisation in Isabelle and Properties

This section outlines the mechanisation of our component semantics in Isabelle/HOL including eager message-based strategy, and several formalised proofs. We have already seen the definition of the component structure in Chapter 5 and now we present the component semantics, directly translated from the preceding sections. We also describe some properties we have already proved using our formalisation, showing that our formalisation is able to handle mechanised proofs entailing reasoning on components, their structure, and first class futures. While the formalisation represents a few hundreds lines of code, the proofs are much longer (above 5000 lines) and entail reasoning interleaving component structure, semantics, and future registration aspects.

6.3.1 Semantics

The semantics of primitive and composite components, as detailed in Section 6.2.2 has been entirely specified in Isabelle/HOL. The various semantic rules presented for composite and primitive components are grouped under the different inductive definitions, corresponding to the four differed reductions presented above (global reduction $\rightsquigarrow$, r-reduction $\rightarrow_R$, and parametrised reductions $\rightarrow_O$ and $\rightarrow_F$).

```
inductive comm::[Component, Component, Name, Fid, Value, Component]⇒ bool
 (_ ⊢ _ -[_ ,_ ,_ ] ↦O _  50)   where
 Call: ⟦ (PintState  s,Call i1 v f,s2) ∈ (Behaviour s);
                                        f ∉ (set (RqIdList S)) ⟧
⟹ S⊢(Primitive N itf s)⊣i1,f,v↦O Primitive N itf (⦇ PintState:=s2 ⦈)|

CompositeCall:⟦ Cqueue s= R#Q; (kind(the(itf (invokedItf R)))= Client;
                f∉(set (RqIdList S)) ⟧
⟹S⊢(Composite N itf subCp bindingss)
                ⊣(invokedItf R),f,(parameter R)↦O
                                     Composite N itf subCp bindings
 (s⦇Cqueue:=Q,CcomputedResults:=CcomputedResults s@
                                        [⦇fid=id R,fValue=(0,[f])⦈]⦈)
```

To compare semantic specification in Isabelle to its mathematical equivalent, we show the Isabelle/HOL version of some semantic rules. The inductive definitions of $\rightarrow_O$ reduction, consisting of Call rule for primitive component and Composite-Call rule for the composite components is shown above. We start with the definition of $\rightarrow_O$ and specify its structure; `(_⊢_ -[_ ,_ _ ]↦O_ 50)` which is same as the mathematical notation for parametrised $\rightarrow_O$ reduction, $S \vdash C \dashv itf, f, v \mapsto_O C'$. Here S is of type component and gives the context, then comes C the component to be reduced. This is followed by the three reduction parameters : (interface name, future, and value). Finally we have the reduced component C'. It is easy to see the equivalence of the two specifications. Only a few intermediate variables and operations were removed/changed in the Isabelle version. For example, `f ∉(set (RqIdList S))` is simplified as $f \notin RqIdSet(S)$.

Similarly, the CommParent rule in Isabelle/HOL is shown below and can be compared with its mathematical equivalent in Figure 6.7. Our mathematical notion however, is easier to follow as compared to Isabelle/HOL formulas. It also allows us to avoid the Isabelle/HOL implementation issues and models our data structures in a formal language independent manner. For example, as already discussed, our definition of RL is semantically closer to mathematical notation one could use for such a data structure, than its Isabelle/HOL alternative (also correct) which relies on lists instead of finite sets (or maps) due to possibility of inductive reasoning on lists. Additionally, we are able to simplify our notation by using set comprehension, which is not supported for lists. We can write $[f, N] \# [(f'', N_0) \mid f'' \in RefFutSet(v)]$ instead of `(f,N) # (map (λid. (id,N0 )) (snd (v))`, for building the list of futures to be registered.

```
CommParent:
 ⟦(subCp ^ N) = Some C, ⦇src=N.itf',dest = This.itf⦈ ∈ bindings;
                                        snd f= N0; S ⊢ C⊣il,f,v↦O C'⟧
  ⟹  S ⊢ (Composite N0 itfs subCp bindings s) →R
              (Composite N0 Itf (subCp<-C') bindings s)
               ← ⦇id=f, param= v, invokedItf= itf ⦈,
                         (f,N) # (map (λ id. (id,N0 )) (snd (v)) |
```

6.3.2 Properties and Proofs on Eager message-based Strategy

The formalisation sketched above and entirely written in Isabelle/HOL is rich enough to allow proofs of various lemmas. Our objective is to have a framework rich enough to address most aspects of distributed components features, but also the framework should be close enough to the existing component framework so that equivalence between the implementation of the framework and the specification is simple and convincing. We believe that our approach is adequate to prove properties entailing component structures, asynchronous communications, and component behaviours. Here, we focus on the formalisation of a future update strategy; we selected eager message-based strategy for formalisation here as discussed in Section 6.1. Consequently, we only present below theorems related to future updates and registration of futures in eager message-based strategy. Of course those properties rely on numerous other lemmas mainly related to component structure, and navigation inside component hierarchy, presented in the previous chapter describing our reasoning framework. Most of the lemmas are proved by induction on the component structure or on the reduction rules.

A first crucial theorem we proved is `UpdatedFutureDisappear`; it assures that when a future has been updated, no reference to this future exist in the updated component. More precisely, when the future `f` is updated at the component with the name `N` inside the component system `S`, the new component `C` (with the name `N`) inside the reduced system `S2` no longer has a reference to future `f`. As already shown, `LocalReferencedRqs` returns the list of futures referenced locally by C; it is similar to *RefFutSet* but does not enter subcomponents. Recall that ˆˆ corresponds to `getRecSubCp` function and looks for the subcomponent with the name N recursively in a component hierarchy, in this case the component system $S2$. The hypothesis `f ∉set (snd v)` captures a crucial requirement for this property; there should not be any future cycles. More precisely, the new value v for the future f does not contain a reference to f itself or simply put, there are no future loops. As shown in Appendix A.8 of [7] in case of cycles, the future update may not terminate, thus making any reasoning quite complex. For more details on future loops refer to [7].

Theorem 6.3.1 (Updated Future Disappears) *Future update removes all references to a given future.*

```
theorem UpdatedFutureDisappear:
⟦S⊣f,v,N↦F S2, RL; CorrectComponent S;(S2^^N) = Some C;  f ∉ set (snd v)⟧
⟹ f ∉ LocalReferencedRqs C)
```

The proof for UpdatedFutureDisappear is approximately 80 lines, and relies on a number of simpler lemmas (not included in count) on local update on values and futures. For example:

```
lemma UpdateFutureSet:
   set (snd (UpdateFuture f v v')) ⊆ set (snd v) ∪ set (snd v')
lemma UpdFut_futdisappear:
   f∉ set (snd v) ⟹ f ∉ set (snd (UpdateFuture f v v'))
```

The lemma UpdateFutureSet simply recalls that once the the current value v' for a future f is updated with the new value v, the referenced futures in the final value are a subset of the the referenced futures inside the two values combined. The future update does not introduce any future outside those that are already known in either v or v'. UpdFut_futdisappear on the other hand implies that after the update, any future that is not referenced inside the value v disappears. Recall that UpdateFutures f v v' replaces the future id f with value v in value v'.

Concerning future registration, the main theorem we proved in Isabelle is the following one :

Theorem 6.3.2 (Globally Registered Futures) *For a correct component system, the global reduction maintains complete future registration.*

```
theorem FuturesRegistered:
 ⟦⊢C1⇝C2; CorrectComponent C1; GlobalRegisteredFuturesComp C1⟧
 ⟹ GlobalRegisteredFuturesComp C2
```

GlobalRegisteredFuturesComp, shown in Section 5.3.2 (under *Future registration*) checks that all futures are registered in the given component system. It states that after global reduction ⊢C1 ⇝ C2, all futures registered in C1 are also registered in the reduced system C2 along with any new future generated as the result of component communications. Although, the proof for this theorem is approximately 70 lines, the proof relies on lemmas about transmission of registered futures, and registration of newly created futures, which are proved separately. Most of those lemma entail much longer proofs and some of them are shown below.

First we show some properties on the underlying reductions that are needed before any proofs on future registrations. Those properties ensure that the various reductions used in our semantics maintain some key constraints. For example, R_maintains_name and O_maintains_name verifies that the component keep it's name after $\rightarrow_R$ and $\rightarrow_O$ reduction. Similar lemma exists for $\rightarrow_F$ reduction.

```
lemma R_maintains_name: S ⊢c1 →R c2, RL ⟹ getName c2 = getName c1
lemma O_maintains_name: ⊢c1⊣il,i,v↦O c2⟹getName c2=getName c1
lemma R_names_eq: ⟦ S ⊢ c1 →R c2,RL; CorrectComponentWeak c1 ⟧
  ⟹ getName '(cpSet c2 )= getName ' (cpSet c1)
```

The proof for R_names_eq is approximately 40 lines and verifies that the component names involved in the composition are the same before and after the reduction.

Using the above mentioned lemmas (and similar ones on other properties) we can prove lemmas on future transmission which are required for our theorem `FuturesRegistered`. The `R_maintainsRegFutures` states that, if a future `f` (in component named `N`) is registered in `C`, and `C` reduces by $\rightarrow_R$ to `C'`, then the `f` is also registered in `C'`. `RegisteredFuture` is shown in Section 5.3.2 (under *Future registration*).

```
lemma R_maintainsRegFutures:
  ⟦S⊢ C →R C',RL; CorrectComponent C; RegisteredFuture f N C; C∈cpSet S⟧
   ⟹ RegisteredFuture f N C'
```

Similarly, the lemma `O_maintainsRegFutures` states that, if a future `fa` is registered in component `C` in the component system `S`, then it is registered in `C'` after emitting the message (and reducing to C' subsequently).

```
lemma O_maintainsRegFutures:
 ⟦ S ⊢ C ⊣il,f,v↦O C';  RegisteredFuture fa N C ⟧
 ⟹  RegisteredFuture fa N C'
```

The next lemma concerns registration of new futures and is a crucial lemma for proving the theorem `FuturesRegistered`. It states that : if in a source configuration, all futures contained in a subcomponent of `C1` are registered in `S` (expressed by `LocalRegisteredFuturesComp`, shown in Section 5.3.2) and the component $C1$ reduces to $C2$, then a future referenced from a subcomponent `C'` of `C2` is either initially registered in `S` or will be registered because a corresponding entry is in the registration list `RL` (Recall that RL stores the registrations required for eager message-based strategy).

```
lemma registeredFutures_R:
⟦S ⊢ C1 →R C2,RL; C1∈cpSet S;∀C∈cpSet C1. LocalRegisteredFuturesComp C S;
     C'∈cpSet C2; f∈LocalRefFutSet C'⟧
⟹ RegisteredFuture f (getName C') S ∨ (f,getName C')∈set RL
```

The proof of the above lemma is more than 200 lines, and itself relies on a number of lemmas, including a similar lemma for $\rightarrow_O$ reduction.

```
lemma registeredFutures_O:
 ⟦ S ⊢ C1 ⊣itf,f,v↦O  C2 ; C1 ∈ cpSet S;
  ∀ C ∈ cpSet C1. LocalRegisteredFuturesComp C S;
  ∀ C' ∈ cpSet C2. ∀ f'. (f'∈  LocalReferencedRqs C') ⟧
 ⟹ (RegisteredFuture f' (getName C') S ∨ (f=f' ∧ C'=C2))
```

Above proofs are almost entirely mechanised: only properties ensuring preservation of `CorrectComponentWeak` by the reductions are left for future works. Informally, it can still be shown why weak correctness should hold for the presented reductions. Recall that `CorrectComponentWeak` requires that the component should have correct structure, all subcomponents (`cpList`) of the component have distinct

names and all the request identifiers are distinct. Taking the example of $\rightarrow_O$, the `O_maintains_name` lemma along with `O_names_eq` and other similar lemmas ensure that component names after the reduction are also distinct (they are distinct before due to `CorrectComponentWeak` constraint).

```
axioms R_maintains_WF:
 ⟦S⊢ C→R C',RL ; CorrectComponentWeak C ⟧ ⟹ CorrectComponentWeak C'
axioms O_maintains_WF:
 ⟦S⊢ C⊣itf,f,v↦O C'; CorrectComponentWeak C ⟧
                                          ⟹ CorrectComponentWeak C'
axioms F_maintains_WF_:
 ⟦S⊣f,v,N↦F S,RL; CorrectComponentWeak C⟧ ⟹ CorrectComponentWeak C'
```

Similarly, we have lemmas (`O_sameRqIdList`) which verify that the reduction preserves the `RqIdList`. Consequently, if the weak correctness holds before the reduction, then in the reduced component the request identifiers are also distinct. Same reasoning applies to the interfaces through `comm_maintains_itfs` and similar lemmas.

```
 lemma O_sameRqIdList:
    S ⊢ C ⊣il,f,v↦O  C' ⟹ (RqIdList C') =  (RqIdList C)
 lemma comm_maintains_itfs:
    S⊢C⊣il,f,v↦O C'⟹ (getItfs C') N =  (getItfs C) N
```

In the same way, we can reason on correct referencing of requests and correctness of component bindings with are required for correct component structure. We hope to formally prove the correctness properties for our reductions in future. For now, as seen above, we use them as axioms.

The theorems and lemmas presented in this section ensure that the eager message-based future update strategy is complete, that is it keep track of the future references in all the component system, and then it updates all those references, removing all references to the considered futures. Consequently, the future can safely be garbage collected. This strategy can be thus adopted in the implementation of the GCM; this guarantees safety of the future update implementation. Although we have only presented our formal semantics with eager message-based future update strategy, other strategies can also be easily formalised. The semantics of our components with lazy message-based strategy appear in Appendix B. Due to the similarities between the two strategy we are quite confident that most of the proofs on basic properties can be easily modified for lazy strategy. Eventually, it could be interesting to prove the semantic equivalence of the three strategies. Indeed the confluence property ASP [7] ensures that that in the absence of deadlocks the results produced by all three strategies are semantically equivalent. We believe that it should be possible to show similar results for our formalisation. Additionally, for the most part ASP proofs were done on paper, while we would have a mechanised version of each proof. Further discussion is provided in Chapter 9.

Chapter 7

Positioning and Concluding Remarks on Formalisation

Chapters 5 and 6 presented a model for distributed components communicating asynchronously using futures. The component model is a subset of GCM model with hierarchical components. The components communicate using an asynchronous request-reply paradigm with futures, where requests are enqueued atomically at the target component and the invoker receives a future representing the result. Futures are first class and consequently future references can spread across components. When the results are available, they are sent to the relevant components using a future update strategy. Three of those strategies are studied in Chapter 3. In this part, we formalised a component model with a full specification of a given future update strategy.

Formalised models. We formalise a subset of GCM model, a European standard for grid component models. To the best of our knowledge ours is the only work focusing on formalising (part of) GCM model. As stated before, our goal is to study the interplay between components and futures. Consequently, our formalisation takes futures into account.

Components in our formalised component model derive most of their structure from GCM. At our level of abstraction, this structure is shared by several component models like Fractal, GCM, and SCA (presented in Section 2.5). However most implementations of SCA (except FraSCAti) do not instantiate the component structure at runtime. By contrast, to allow component introspection and reconfiguration at runtime, we consider a specification where structural information is still available at runtime. This enables adaptive and autonomic component behaviours. Indeed, component adaptation in those models can be expressed by reconfiguration of the component structure. For example, reconfiguration allows replacement of an existing component by a new one, which is impossible or very difficult to handle in a model where component structure disappears at runtime. FraSCAti [72] is an implementation of the SCA model built upon Fractal making this implementation close to GCM. It provides dynamic reconfiguration of SCA component assemblies, a binding factory, a transaction service, and a deployment engine of autonomous SCA architecture. Due to the similarity between FraSCAti and GCM, our approach provides a good formalisation of FraSCAti implementation.

ProActive/GCM, a reference implementation of GCM model, comes closest to our formalised component model. ProActive/GCM may be considered one possible

implementation of our formalised model. Table 7.1 highlights some of the differences between the GCM specification, its reference implementation ProActive/GCM and our formalised model. As seen in the table, GCM specification allows for a lot of flexibility, for example the specification does not impose any particular concurrency or communication model. ProActive/GCM only implements mono-threaded components. Each component in ProActive/GCM has a single execution thread with serves the requests. Our formalisation allows concurrent execution of requests. Similarly, GCM allows for a variety of communication models, synchronous, asynchronous, event-based, etc. For reasoning on components, a more precise formal specification is required. Consequently, we restrict the communication model to an asynchronous request-reply paradigm with futures. All communication between components is in the form of asynchronous method invocations with futures as placeholders for result values. Communication in ProActive/GCM takes place using the same communication model. In contrast to GCM and ProActive/GCM, for now all bindings in our formalised model are one to one (it is still possible to have several bindings reaching the same server interface); all interfaces are unicast interfaces and do not support group communication. GCM and ProActive/GCM supports specification/implementation of functional and non-functional interfaces. However, in our formalisation we focus solely on functional interfaces and do not model non-functional interfaces. We also do not have the notion of optional interfaces, which are allowed by the specification. There are no controllers or non-functional interfaces in our component structure. ProActive/GCM provides a detailed process for component deployment through ProActive deployment framework [15]. Our formalisation does not cover deployment.

Most existing works on formal methods for components focus on the support for application development whereas we focus on the support for the design and implementation of component models themselves. To our knowledge, this work is the only one to support the design of component models in a theorem prover. It allows proving very generic and varying properties ranging from structural aspects to component semantics and component adaptation.

Finally, there is no formalisation available for GCM specification. In [88, 89], authors focus on verification of behaviour of GCM components. However, they prove properties of specific applications rather than provide a formalisation of the component model itself. [82] comes closes and provides a comprehensive formalisation of Fractal component model specification, from which GCM components inherit most of their structure.

Summary of contributions A first formalisation of GCM model and its runtime semantics appear in [1]. Our work is an extension of the model presented there, and we focus on formalisation of the component model and its runtime semantics in the presence of a future update strategy.

Chapter 5 presents our framework for supporting mechanised proofs for distributed components, formalised in Isabelle/HOL theorem prover. In particular we

Features of Component Models			
Property	**GCM**	**ProActive/GCM**	**Formalised model**
Hierarchy	Yes	Yes	Yes
Distribution	Primitive/ composite	Primitive/ composite	Primitive/ composite
Concurrent execution	Un-specified	Mono-threaded	Potentially multi-threaded
Group Communication	Yes	Yes	No
Communication paradigm	Un-specified	asynchronous requests with futures	asynchronous requests with futures
Functional interfaces	Yes	Yes	Yes
Non-functional interfaces	Yes	Yes	No
Non-functional components	Yes	Yes	No
Formal semantics	No	Partial	Yes
Deployment	High level	Yes	No
Optional interfaces	Yes	Yes	No

Table 7.1: Properties of our formalised model vs ProActive/GCM

focus on the handling of component structure, on a basic set of lemmas providing valuable tooling for further proof, and the illustration of the presented framework to prove a few properties dealing with component semantics and reconfiguration. We present the logical machinery of a mechanised framework for reasoning about structured component systems; especially targeting distributed components with futures. We have first illustrated and motivated the specification of components and the provided proof infrastructure. Furthermore, we have shown this machinery in action by showing how reconfiguration of components can be formally specified, and how properties over component structure and reconfiguration can be handled.

Chapter 6 relies on the reasoning framework built in Chapter 5 and presents the runtime semantics of our components, incorporating one future update strategy. Future update strategies are somewhat neglected in the literature. We believe that even though future update strategies need not be included for studying properties of a language, they are still important for reasoning on the implementation of this language. Consequently, our semantics include formalisation of one future update strategy, the eager message-based strategy. We also sketch the first proofs and supporting lemmas related to properties on future registrations. Our model is precise and expressive enough to reason about futures and components, and to guarantee correctness properties. Component semantics for a second future update strategy, the lazy message-based appear in Appendix B, demonstrating that our formalisation

is flexible enough to support multiple future update strategies.

All of our work, the component model specification, its semantics, and proofs of properties, has been mechanised in Isabelle/HOL theorem prover. Those mechanised proofs ensure the correctness of the implementation of future updates in ProActive/GCM. Overall, the developed framework consists of more than 4000 lines, including almost 300 lemmas and theorems, approximately 500 lines for defining the component model and its semantics, and 1800 lines focusing on properties specific to future update strategy. The remaining code proves auxiliary lemmas and general properties on the component structure.

Chapter 8

Conclusion

A natural way to benefit from distribution is via asynchronous invocations to methods or services. Upon invocation, a request is enqueued at the destination side and the caller can continue its execution. But a question remains: *what if one wants to manipulate the result of an asynchronous invocation?* First class futures provide a transparent and easy-to-program answer: a future acts as the placeholder for the result of an asynchronous invocation and can be safely transmitted between communicating remote processes while its result is not needed. Synchronisation occurs automatically upon an access to the result. As references to futures disseminate, a strategy is necessary to propagate the result of each request to the process that needs it. In this thesis, we studied the first class futures focusing on the mechanisms for transmitting the results for futures; the *future update strategies*.

The first part of the thesis provided a detailed semi-formal specification of three main future update strategies adapted from [7]; we then used this specification for implementing the strategies in a distributed programming library. We studied the efficiency of the three update strategies through experiments. The semi-formal specifications and the experimental results are published in [9, 10].

The second half of the thesis dealt with more formal aspects. Proving correctness of distributed protocols has always been a challenging task. In our opinion, modern theorem provers are sufficiently advanced to make mechanised reasoning feasible. Mechanised proofs remove the chances for human errors that may occur in the traditional pen-and-pencil proofs. In particular they remove the possibility of proving wrong statements given that the underlying logic is consistent and valid.

We presented a model for hierarchical distributed components communicating asynchronously using futures. The communication model is based on a request-reply paradigm, where requests are enqueued at target component and invoker receives a future, representing the result. Futures are first class: and consequently future references can be spread across components. When the results are available, they are sent to the relevant components using a future update strategy. Our component model, its semantics, and proofs on properties are all formalised in Isabelle/HOL theorem prover. Our presented formalisation and the proof of various properties are published in [13, 14].

Contributions, results, and impact

The contributions of this thesis include:

A Generic semi-formal notation

We introduced a generic and language independent notation for modelling the future update protocols. Our chosen notation interprets the future update strategies as combination of *events* and support *operations*. Support operations simplify the task of specifying the details of different strategies. For example, *register futures, local update of futures*, and *garbage collection of futures* are all support operations. Future update strategies react to various events/stimuli, triggered by application or the middleware. Our approach allows us to model the actions taken by a future update strategy to ensure proper response to a particular event/occurrence. For example, *creation of a future, transmission of a future, computation of the value for a future*, etc., are all events in the life-cycle of a future. Decoupling the notation from real implementation allows us to specify future update strategies in a generic manner; the resulting specification can be used for other works using first class futures, for example Creol and AmbientTalk, as well.

Semi-formal specification of update protocols

We used our generic semi-formal notation to specify three future update strategies; two eager strategies (eager forward-based and eager message-based) and one lazy strategy (lazy message-based) were presented. A higher-level view of these strategies appeared in [7]. However, in comparison to our work, [7] is more abstract and does not sketch the working of each strategy and how it may be implemented. In contrast, our specification is more detailed (and precise) and clearly specifies the underlying mechanisms and data-structures required for implementing the strategies. For example, we show *when should a process register for a future*, and *which data-structures* ($\mathcal{FR}$ and $\mathcal{FL}$) are needed to keep track of futures. Our Semi-formal specification is precise-enough and contain sufficient details to be used – and has been used – as basis for a real implementation. We believe that such semi-formal approaches strike a good balance between the ambiguities inherent in informal descriptions and the complexities of formal mathematical notations. Finally, even-though we presented three future update strategies, in our opinion, the presented approach is flexible enough to model other strategies that may be envisioned.

Analysis of future update strategies

Availability of a detailed specification of future update protocols, allowed us to *informally estimate* the efficiency of strategies in terms of *message exchanges*, and *time to update futures*. We showed a basic cost-analysis model for our selected strategies to understand the costs and trade-offs required by each strategy. We estimated the costs using indicators such as *total number of messages exchanged, time to update a given future*, and *time to transmit a given result to all processes with corresponding future*. We analysed those indicators on parameters such as *number of intermediate hops/processes*, *possible number of concurrent updates*, and *the time required for serialising the payload*. The goal was to provide directions on a possible way of analysing the strategies;

we consider a detailed (and consequently more complex) cost model to be out of scope of this thesis.

Implementation of missing strategies

We presented our implementation of future update strategies in ProActive middleware – a distributed programming library based on ASP-calculus. By default, ProActive provides two modes: *no automatic continuation mode* (configured in ProActive descriptor) disables the use of first class futures. Alternatively, the programmer can decide to use first class futures, which are supported using *eager forward-based* strategy. We extended the support for first class futures in ProActive and implemented *eager message-based* and *lazy message-based* strategies. Our implementation builds on the work presented in [16], and provides support for *nested futures*, newer configuration options, along with numerous other modifications necessary to work with new versions of ProActive. In addition to the default options of ProActive, we now offer two additional modes corresponding to the two additional strategies; these may be configured via an XML descriptor. While our implementation of update strategies is not optimised, we believe that it provides a good starting point for studying the behaviour of various future update strategies.

Experiments on future update strategies

To validate the implementation and to further improve our analysis, we presented results for two experiments using different future update strategies. Our experiments were carried out in collaboration with authors of [16], and together with our analysis on future updates, strove to answer the non-trivial question: *Which is the best future update strategy?* While the presented experiments are not sufficient to settle this question, they demonstrate that the performance of a strategy can greatly vary depending on the nature of the application (and with the previously presented parameters). Lazy strategy is most suited in situations when only some of the processes with the future actually require the result value. On the other hand, lazy strategy also introduces an additional time delay before the result can be acquired and as such may not be suited for applications which require result values to be transmitted as soon as they become available. The single step update characteristic of message-based strategies reduces the time delay as results are serialised only once. However, compared to eager forward-based strategy, they consume more bandwidth and are more memory intensive. We believe that the presented results justify our decision of studying various future update mechanisms and sufficiently demonstrates that *there is no single best strategy suited to all scenarios*. Rather, applications can benefit from having more than one supported strategy. A large scale experimental study would be instrumental in analysing the various future update strategies; however we consider such a study to be out of scope of this thesis.

A formalised distributed component model

We formalised in Isabelle/HOL [12], a subset of GCM model – a European standard for grid component models. Our formalisation includes distributed components, futures and asynchronous communications by means of requests. Hierarchy is modelled as follows: our components can be *primitive* or *composite*, primitive components are leaf-level components and implement the business logic; Composite components compose one or more subcomponents. Each component has a unique *Name*, a list of component *interfaces*, and a component *state*. All communications between the components is via their *public interfaces* (uni-cast). Components receive incoming messages on *server interfaces* and emit messages (invoke method-calls) on *client* interfaces. Composite components have, in addition, a list of *subcomponents*, and a set of component *bindings*. The asynchronous communication is supported through first class futures. Incoming requests are enqueued atomically in the *message queue*, while the invoker receives a future. As opposed to GCM specification, our formalisation does not include non-functional interfaces. All our interfaces are uni-cast, and we do not support multi-cast or gather-cast interfaces.

The behaviour of primitive components is governed by a LTS *behaviour*, the possible internal actions are: internal transition, request service, request emission, result reception, and end of service which associates a result to a request. In comparison, the behaviour of composite components is quite limited. Essentially, a composite component serves the requests in a FIFO order, and delegates all incoming requests to its bound components. A request is emitted by a client interface of a primitive component, and received unchanged by the server interface of the primitive component that is (indirectly) bound to it. The delegated requests arrive at the bound component unchanged and may traverse several composite components and bindings.

To the best of our knowledge ours is the only work focusing on formalising (subset of) GCM model. The GCM reference implementation ProActive/GCM can be considered as a possible implementation of our formalised component model.

Infrastructure for efficient manipulation of component structure

Component models ensure that components have a well-determined structure which facilitates reasoning on component structure and interactions. To take advantage of this strict adherence to the structure and to manipulate composition hierarchy, we built an infrastructure for handling components in Isabelle/HOL. Various operations that allow us to effectively manipulate components were shown; those include: operations for *accessing component state*, mechanisms for *traversing component hierarchies*, and means for *replacing* and *updating components* inside the hierarchical structure. For example, we presented *cpList* construct that gives a list of all subcomponents of a component recursively. Operations like *getRecSubCp*, and *changeCp*, etc., were shown to demonstrate how we manipulate the components inside a composition hierar-

chy. Similarly, we have operations for manipulating requests and computed results, needed for reasoning on correctness of update strategies. For example, the operation *ComputedRqs* allows us to retrieve all computed requests in all subcomponents of a composite. All these operations are primitive recursive functions enabling an encoding in Isabelle/HOL using the `primrec` construct. Using this construct has great advantages for the automation of interactive reasoning process. Automated proof procedures of Isabelle/HOL, like the simplifier, are automatically adapted to new equations; simple cases can be solved automatically. Moreover, the definitions themselves must use pattern matching leading to readable definitions.

Properties on correct components

We specified the structure of *well-formed* and *correct* components in Section 5.3.4. We only reason on a subset of all possible components (which meets these two criteria) that can be constructed according to the described component structure. A composite component is well-formed (has correct structure) if : each binding connects an existing client interface to another existing server interface; each client interface is connected only once; all subcomponents have a distinct name; and all requests in the request queue of the composite refer to existing server interfaces. A primitive component has a correct structure if it follows the last requirement plus some additional constraints relating its behaviour with its interfaces.

A *correct component* is a well-formed component that also has uniquely defined request identifiers (recursively), and all future referenced by the components should correspond to an existing request. Finally, names of all components in the composition should be unique. This differs from the well-formedness requirement which only requires the names of all direct subcomponents to be unique.

The requirement of checking correct future referencing throughout the composition hierarchy is stronger than what is needed for most proofs, and can at times be relaxed resulting in a *weak correctness* requirement. Weak correctness eases proofs involving component hierarchy because if a component verifies weak correctness, then all its subcomponents also verify it; which is not the case for *correct component*.

We proved properties on the correctness of component structure. The properties logically relate the degree of correctness of the structure. We presented some of those lemmas. For example, we showed lemmas that established the well-formedness of the subcomponents of a well-formed composite component. Another lemma established the well-formedness of all constituent components. Other presented properties included: properties relating *weak correct components* with *correct components*, verifying uniqueness of components (w.r.t *names*), etc.

As a consequence of the mapping between component structure and Isabelle's

structural support, it has been relatively easy to prove properties of component structure by automatic steps plus induction on the component structure.

Runtime handling of components reconfiguration

Reconfiguration represents all the transformations of the component structure or content that can be handled at runtime. We considered only structural reconfiguration, which includes changes of the bindings, and of the content of a component. Our framework includes the structural information in the formalisation. This not only allows us to efficiently manipulate and reason on component structure, but also enables reasoning on reconfiguration primitives and behaviour of a reconfigured component system. We showed two reconfiguration primitives : *unbind* removes one binding inside a composite component, while *replace* changes the subcomponent of a composite component. We defined a *completeness* property for our composites. A composite component is complete if all interfaces of its subcomponents and all its internal interfaces are bound (note that we do not have optional interfaces). Using this definition of completeness (derived from Fractal), we showed that our *replace component* primitive respects the completeness constraint.

Component configuration and reconfiguration is a vast topic, and is not the goal of this thesis. The formalisation of reconfiguration primitives and the properties provided here only serve to show that our framework is detailed enough to allow reasoning on component configuration-reconfiguration.

Runtime semantics with eager message-based strategy

We presented a runtime semantics for our components; our semantics incorporate formalisation of one future update strategy. Formalising future updates is of little interest concerning the language properties, but it is crucial to study the implementation of this language. In order to prove the correctness of the implementation of GCM, our work aimed at specifying formally future update strategies and proving correctness properties on futures.

Based on the presented component structure, we can derive semantics using any of the previously mentioned strategies; all three strategies are semantically equivalent under a *decidability hypothesis*, as demonstrated in ASP-calculus [7]. Of the three presented future update strategies, eager message-based strategy is the most complex strategy and consequently, we selected it for our formalisation and proofs. Semantics of a second strategy (lazy message-based) were also shown in an Appendix, demonstrating the flexibility of our approach. Our semantics are fully formalised in Isabelle/HOL, together with the proof of properties.

We define the runtime semantics of our components by a number of inductively defined reduction relations. The global reduction of the component system is $\rightsquigarrow$, which triggers either $\dashv f, v, N \mapsto_F$, or $\rightarrow_R$ reductions. Also a parametrised reduction relation $\dashv \mathit{itf}, f, v \mapsto_O$ (interface name, future, value) emits messages and is used by $\rightarrow_R$ reduction. The parametrised relation $\dashv f, v, N \mapsto_F$ (future,

value, component name) expresses future updates. Using these reduction rules, we provided first the semantic rules for primitive components, followed up by the semantic rules for composite components. Both semantics incorporate the formalisation of eager message-based future update strategy. Finally, we give the semantics for triggering the future update globally and the semantics for our global reduction $\rightsquigarrow$, which ensures that relevant future registrations are performed; this completes our semantics for GCM-like components with eager message-based future update strategy.

Correctness of future updates

Our presented formalisation in Isabelle/HOL is rich enough to allow proofs of various lemmas. Our objective was to have a framework rich enough to address most aspects of distributed components features; but also the framework should be close enough to the existing component framework so that equivalence between the implementation of the framework and the specification is simple and convincing. We believe that our approach is adequate to prove properties entailing component structures, asynchronous communications, and component behaviours. We focused on the formalisation of a future update strategy; we selected eager message-based strategy for formalisation here as already discussed. Consequently, we only presented theorems related to future updates and registration of futures in eager message-based strategy. Of course those properties rely on numerous other lemmas mainly related to component structure, and navigation inside component hierarchy. Most of the lemmas are proved by induction on the component structure or on the reduction rules.

We have proved two main theorems concerning futures. The first theorem ensures the correctness of future update operation and verifies that *a future update removes all references to a given future*. The second theorem is the main property that we prove on future registrations and establishes the correctness of formalised future update protocol; the theorem verifies that *for a correct component system, the global reduction maintains complete registration of futures*. These two theorems (and the other presented lemmas) ensure that the eager message future update strategy is complete, that is it keep track of the future references in all the component system, and then it updates all those references, removing all references to the considered futures. Consequently, the future can safely be garbage collected. This strategy can be thus adopted in the implementation of the GCM; it guarantees safety of the future update implementation.

The proofs of the above theorems are almost entirely mechanised: only properties ensuring preservation of *correct component structure* by the reduction rules are left for future works. Informally, we sketched in Section 6.3.2 how a possible proof can be constructed.

8.1 Final remarks

This thesis focused on a study of first class futures, in particular on the mechanisms for transmitting the result values. Future update strategies are somewhat neglected in the literature. We believe that even though future update strategies need not be included for studying properties of a language, they are still important for reasoning on the implementation of this language. The work encompassed both applied and formal aspects. We presented a semi-formal specification for future update protocols. Based on the semi-formal specification, we implemented the future update strategies in ProActive and performed some experiments to validate our strategies. We believe that even without a large scale experimental study, the results are sufficient to justify the need for having more than one future update mechanisms. Our implementation provides a good starting point for studying and analysing the performance of various strategies. Although, the implementation is in one particular middle-ware, it is based on a language independent specification. As a result, we believe that the results presented here can be applied to other frameworks that make use of first class futures as well.

The formal part of the thesis presented the logical machinery of a mechanised framework for reasoning about structured component systems; especially targeting distributed components. We first illustrated and motivated the specification of components and the provided proof infrastructure. Furthermore, we have shown this machinery in action by showing how reconfiguration of components can be formally specified, and how properties over component structure and reconfiguration can be handled. We also illustrated our approach by showing the specification of a semantics for components, and associated proofs. Throughout our formalisation process, we provide a justification and reasoning behind our design choices.

Our approach focuses on increasing confidence in global properties of component models. For this, we provide a framework and apply it to prove generally valid results. The established infrastructure of structured components with asynchronous communication provides an elegant abstraction from implementation details while fully preserving the communication structure, and defining a precise semantics that is required for mechanised reasoning. We provide support for distributed components communicating by asynchronous requests with futures. Overall we have developed a reliable basis for the mechanical proofs of properties of hierarchical component models, and we have shown its adequacy to deal with first proofs entailing reconfiguration, and component semantics.

We now have sufficient formal constructs and tools to express future update strategies and to study their properties. This work showed that it is possible to formally prove completeness and correctness of our future update mechanism, and of the corresponding implementation in ProActive/GCM. A crucial point during the specification phase was to find the good Isabelle/HOL abstraction to represent the component structures. We think we found a good balance between expressiveness and abstraction, that allows formal reasoning on the interplay between the execution and component structure, but is close enough to the component model

implementation.

Finally, the work presented in this thesis is by no means complete. We hope that it leads to further discussion and study of first class futures, in both the applied and theoretical fields. We discuss some of the future possibilities in the next and final section of the thesis.

Chapter 9

Future Works

As discussed in relevant sections, this work is not complete. Rather we hope to provide a strong basis for further research on future update protocols. In the following sections, we present some perspectives for future efforts; we discuss some short term and long term future perspectives. Following the pattern of the thesis, we discuss separately the directions for the formal and the applied domains.

9.1 Applied Aspects

Our implementation of future update strategies provides a good starting point for further optimisations and experimental evaluation. We discuss some of the possibilities in the following:

Case study to evaluate efficiency of future update strategies

We presented the results from some experiments on evaluating the performance of different strategies. As discussed, those experiments are not sufficient to fully evaluate the efficiency of the presented strategies. Therefore, an important next step is to carry out a large scale experimental study on future updates. The current set of experiments were carried out using a relatively small local cluster. The case study could be executed and analysed in a Grid or Cloud infrastructure. In particular, using Grids with nodes at different physical locations would help measuring the network related issues in more detail. Such a case study would help improve the confidence in the analysis presented in this thesis.

Some possible parameters that could be better explored with a case study are:

Size of the result value: As discussed, size of the result value has important repercussions for all strategies. For eager forward-strategy, results are serialised-deserialised at each intermediate process. To evaluate the impact of the size of the result value on eager forward-based strategy, one could tailor use cases that require transferring large amounts of data as part of future updates. An important improvement would be to separately measure the ratio of time spent in serialisation-deserialisation, and the actual data-transfer.

Memory usage: The message-based strategies in general utilise more memory than eager forward-based strategy; results have to be stored for longer durations. For example, in theory the computed results in lazy message-based

strategy can not be garbage collected. A comparative analysis of memory usage of each strategy could be of interest, particularly for applications where memory could be an issue.

Network bandwidth: Network bandwidth is another property of interest while discussing message-based strategies. The message-based strategies rely on being able to perform all updates in a central manner. This can potentially create a bottleneck, particularly for eager message-based strategy, if sufficient bandwidth is not available for concurrent updates of future values.

We presented a worst-case scenario for eager forward-based future update strategies (future updates along a chain of processes). Similarly, different scenarios could be constructed and evaluated to exploit potential weaknesses of each strategy. For example, lazy message-based strategy can (potentially) perform worse than eager message-based strategy if every process that receives a future, makes a future access. If the future is accessed after computing process has produced a result, then additional delays would be encountered as each process sends a message asking for the result and waits until the result arrive. In contrast, eager message-base strategy would communicate the result as soon as it is computed; result would arrive faster.

Garbage collection

Garbage collection is an important aspect of programming. However, this thesis was oriented towards study of future updates and we did not cover garbage collection of computed results. An interesting follow up to our work could be to implement proper garbage collection mechanisms for our strategies.

For eager forward-based strategy, garbage collecting the computed values is a straight forward process; a result can be removed as soon as it is forwarded to all processes to which the corresponding future was previously communicated.

However, for message-based approaches, garbage collection is a more challenging task. In eager message-based strategy, computed results can only be garbage collected when: Every processes that registered for a particular result, receives the result; and the corresponding future is not in transit. Once these two requirements are met, results may be safely garbage collected. As already discussed, it is easier to check for in-transit futures in a sender-based implementation of registration mechanism.

Similarly, solutions could be explored for garbage collecting the computed results for lazy message-based strategy. In theory, processes could ask for a computed value at any time, thus requiring the processes to hold computed values indefinitely. In practise however, application dependent timeouts could be used for removing results.

A possible way ahead could be as follows:

- Design of a distributed algorithm to check in-transit futures,

- Implementation of mechanisms for allowing the programmer to specify timeout values for computed results,
- Correctness proofs on garbage collection mechanisms.

Formal reasoning on such algorithms is important to guarantee correct operation of the algorithm. For eager message-based strategy, it would assure that there are no futures in-transit. For lazy message-based strategy, in addition, we could have proofs ensuring that the garbage collected future is indeed redundant, and can (will) no longer be accessed by any processes.

An alternative to a distributed garbage collection algorithm could be to allow the programmer to designate one process (preferably with access to high bandwidth connection) as a 'results server'. Then all processes that require a result could be redirected to this result sever. Once results have been communicated to the results server, they could be garbage collected locally. However, the results server would still require a proper garbage collection mechanism.

Optimising the implementation

We provide a real implementation of future update protocols. However, the implementation is not fully optimised and a number of improvements could be suggested.

Eager message-based and lazy message-based strategies both rely upon centralised concurrent updates. Currently, we perform these updates using a thread pool approach. The result value is serialised, and is handed over to a thread pool which can then do concurrent transfers of future value. A better approach could be to use a group-communication API to multicast the produced result. There are a number of such APIs available that provide a simple and optimised solution for communicating data-values to a group of interested processes.

Another long term perspective improvement could be the implementation of an 'un-registering' protocol, allowing a process to declare that it is not interested in the result of a received future anymore. Such a protocol could potentially be quite useful for eager message-based strategy. A similar protocol could be designed to allow eager forward-based strategy to skip some intermediate processes that do not require the results (in case of long process chains). A set of possible steps could be:

- Design of an un-registering algorithm for forward-based and message-based strategies,
- implementation of the algorithm,
- correctness proofs ensuring that the algorithm does not result in futures that cannot be resolved anymore.

Design and implementation of such protocols is a challenging task requiring significant analysis, to ensure that such un-registering does not effect any other

futures. However, if implemented efficiently, it could potentially improve the performance of the implemented strategies. A formal study along side the design and implementation of the algorithm would improve the confidence in the 'correctness' of the algorithm. A correct un-registering algorithm must ensure that there are no futures depending on the arrival of that particular result.

Hybrid strategies

Another challenging follow up could be supporting use of mixed strategies. Currently, we use the same future update strategy for all processes/components. A more interesting approach could be to select a particular strategy on per process or component basis. For example, each future created as a consequence of method-invocation on a particular component would be updated following a certain strategy. In other words, each process or component would decide how it wants to communicate all the results that it computes. Such flexibility would be quite advantageous, as it would allow each process or component to be configured to use the (somewhat) optimal strategy based on the characteristics (data-size for example) of the results it produces. The information on result computation policy (for example, strategy-name and the computing process in case of eager message-based strategy) could then be embedded in the future when it is created. Each component would however need to support all three strategies as the incoming futures could require different strategy.

Mixing strategies in this way can be quite challenging. Each process would need to treat incoming futures differently based on the policy associated with that future. So for one future a process might perform a registration, while for the other the results would arrive via eager forward-based strategy. Such mixing will also greatly increase the complexity of garbage collection algorithm.

An approach can also be envisioned where its the receiving processes that decide how the results should be communicated to them. For example, a time critical process may decide that to minimise the transmission delays, it wants to receive results for all incoming futures in a eager manner (eager message-based strategy would be more suited here). This would force the process producing the result to fine tune its result communication policy for individual futures. An extreme case could be when one process asks for receive the result in an eager manner, but another process will ask for the result using lazy message-based strategy. This kind of coordination between the various processes and garbage collection of such result is indeed be a difficult challenge, requiring considerably amount of further research work.

9.2 Theoretical Aspects

We have formalised a subset of GCM component model and proved some correctness properties on future updates. Our proofs are not exhaustive and some interesting results are left as future work. We discuss some such results in the following:

Reduction rules and component correctness

We present our semantics as a set of inductively defined reduction rules. As we discussed in Section 6.3.2, the presented proofs of properties on future update (with eager message-based strategy) are almost entirely mechanised. The only properties remaining are those that ensure preserving of `CorrectComponentWeak` by the reduction rules. We are confident that these properties are preserved by the reductions; most of the lemmas and properties required for this proof are already proved. We showed an informal sketch of how this proof can be constructed. In short term we intend to complete the mechanised proof on preservation of weak correctness by the reduction rules, hence arriving at fully mechanised proofs.

Correctness properties on lazy message-based strategy

We presented the semantics of lazy message-based strategy in Appendix B. Currently we do not have any proofs on this strategy. In short term, we hope to prove the correctness of lazy message-based update protocol. In particular, we wish to prove that *if the result for a future is computed, and the future is awaited at some component, the component will receive the result.*

Lazy message-based strategy is quite close to eager message-based strategy; as can be observed from comparison of the two semantics. Both strategies utilise similar data structures and rely on a registration mechanism. The main differentiating point between the two semantics is the timing of future registration. In lazy message-based strategy, futures are registered only on a *wait-by-necessity*; registration is triggered on future access. Due to the similarities between the two semantics, we are confident that the basic lemmas (and operations) can be easily adapted to the lazy approach. Once the basic blocks are there, a correctness proof could be constructed for lazy message-based strategy as well. In the short term we intend to focus on this particular proof.

Formalisation of eager forward-based strategy: semantics and proofs

We presented the formal semantics for our components with eager message-based future update strategy (with properties), while the semantics with lazy message-based approach were shown in the Appendix B. While we believe that eager forward-based strategy would be easier to formalise, we currently do not provide a formal semantics for this strategy.

A potential direction could be to formalise the eager forward-based strategy. In particular, such a formalisation would be needed for the eventual long term goal; proving equivalence of future update mechanisms.

Semantic equivalence of future update protocols

Once all three main future update strategies are formalised, a logical next step could be to prove that the three main update protocols are *semantically equivalent*. Work done in ASP-calculus [7] could be used as a potential starting point for establishing such semantic equivalence. Under similar hypothesis, it should be feasible to show that the order of future updates is not important, and all three strategies may yield similar results. Moreover, in our case it would be required to introduce additional hypothesis on the behaviour of each primitive component. Indeed ASP semantics is more precise than the formalisation presented here; we purposely left the behaviour of primitive components underspecified in order to take into account several programming models. Additionally, the properties proved in ASP are for mono-threaded processes, and would require establishing an equivalent specification for our components (our components can have more than one service threads).

Formal semantics and proofs for hybrid strategies

As already discussed, mixing of the three strategies can be a challenging task. A formalisation would be quite helpful in ensuring correct behaviour of the resulting system. However, such a formalisation is a non-trivial task. Indeed for proving properties on such mixed strategies, one would need to either:

- Combine the semantics of the future update strategies. Our presented approach currently favours defining different semantics for each future update strategy. A long term goal is to have semantic equivalence for the presented strategies. However, for proofs on correctness of mixed strategies, this current approach is not sufficient. Instead of having three different semantics, a single semantics with all three strategies is required. From our experience with the semantics of the presented strategies, defining such combined semantics would be a long process. The resulting semantics would contain a lot of complex rules.

Or a second approach could be to:

- Define a parametrised semantics with the parameters identifying the particular strategy. In our opinion, developing such a parametrised semantics will be a very challenging task. Indeed, the design of such a semantics is far from trivial, and reasoning on a parametrised semantics will reveal more complexities.

In either case, this provides some interesting possibilities for future work.

Formalisation of a larger set of reconfiguration primitives

Component reconfiguration is an important research area. We presented the formalisation of two reconfiguration primitives, showing that our framework is expressive enough to support structural reasoning on component reconfiguration. An interesting further development could be to formalise a larger set

of reconfiguration primitives for distributed components. For example in [98], the authors present a reconfiguration primitive for reconfiguring components in a distributed manner, and to allow remote invocation of reconfiguration scripts. Such work could go with the design of a new language for writing reconfiguration procedures. It also takes into account the distributed execution of reconfiguration scripts and the synchronisation between them. A formal representation could help in proving the correctness of reconfiguration and adaption procedures.

Appendix A

Summary of terms and notations

Table A.1: Summary of symbols and notations

symbol and operations	description
Chapter-3	
$\mathcal{A}$	Set of processes(activities): $\alpha, \beta.. \in \mathcal{A}$
$\mathcal{F}$	Set of futures
$f^{\alpha \to \beta}$	Future f created as a result of an asynchronous method call by process α on process β. β will compute the future value.
$\mathcal{FL}_\alpha$	Future list at process α. Keeps track of all referenced futures for this process.
$\mathcal{FR}_\delta$	List of future recipients at process δ. Keeps track of all the processes to which a result should be sent when available.
Reg_δ	Register future operation at process δ. Adds <future, process/location> in $\mathcal{FL}$ or $\mathcal{FR}$.
$Update_\delta(loc, v)$	Future update operation at process δ. Locally updates all occurrences of a given future (at memory position loc) with the result value v.
$Clear_\delta(f^{\alpha \to \beta}, L)$	Remove future operation. Removes all occurrences of a future $f^{\alpha \to \beta}$ from the list L (either $\mathcal{FL}_\delta$ or $\mathcal{FR}_\delta$).
$SendValue_{\delta \to \gamma}(f^{\alpha \to \beta}, v)$	Transmit future value operation. Sends the result value v corresponding to future $f^{\alpha \to \beta}$, from process δ to process γ. The value v may also contain futures.
$Create_\alpha(f^{\alpha \to \beta}, loc)$	Create a new future operation. Creates the result value v corresponding to future $f^{\alpha \to \beta}$, from process δ to process γ.
$SendRef_{\delta \to \gamma}(f^{\alpha \to \beta}, loc)$	Future forwarding operation. Communicates a future $f^{\alpha \to \beta}$, from process δ to process γ.
	Continued on next page

Table A.1 – continued from previous page

symbols and operations	description
$FutureComputed_{\beta}(f^{\alpha \to \beta}, v)$	Request execution termination operation . Occurs when the execution of a request corresponding to future$f^{\alpha \to \beta}$ is completed and a result value v is produced. v may contain other futures.
$Wait_{\alpha}$	Access to an unresolved future. This event occurs when a process attempts to access the value of an unresolved future. The accessing execution thread is blocked until the result arrives.
Chapter-4	
`newActive(...)`	Method for creating a new active object. Requires the *class* of the object.
`turnActive(...)`	Method for turning a normal java object into an active object.
`FuturePool`	Class in ProActive which keeps track of futures. Provides the functionality of $\mathcal{FR}$, $\mathcal{FL}$ list in semi-formal specification.
`FutureMap`	Class in ProActive which implements a mapping between a future and its corresponding automatic continuation (mechanism for updating the results of first class futures).
`ActiveACQueue`	A thread inside an active object for communicating future values.
`RequestForFutures`	A registration message for adding processes as future recipients. Used in message-based strategies.
Chapter-5	
Isabelle/HOL Syntax and Notation	
`HOL`	A theory in Isabelle/HOL theorem prover, encoding the higher order logic.
(a_1, a_2)	Pairs with the datatype $(\tau_1 \times \tau_2)$, where $a_1$has type τ_1 and a_2 has type τ_2.
`nat`	Datatype for natural numbers.
`bool`	Boolean datatype.
#	Operator for constructing lists.
@	List append operation.
`option`	Option datatype in Isabelle/HOL. Allows to cater for an exceptional case for a given datatype.
`datatype`	Isabelle/HOL keyword for defining new (possibly recursive) datatypes .

Continued on next page

Table A.1 – continued from previous page

symbols and operations	description
`List,Set`	Isabelle/HOL theories providing lists and sets (with their related operations and properties) respectively.
`map`	List operation which applies a given function to all elements inside the list.
`x.P`	Set comprehension notation for all elements that satisfy the given predicate `P`. Sets in Isabelle/HOL are typed and all elements of a set share the same datatype τ.
`set`	List operation which returns a set of all elements in a given list. Useful for quantifying over elements of a list.
$\Rightarrow$	Isabelle/HOL notation for specifying a function type. (`nat list`)$\Rightarrow$(`nat set`) is a function that takes a list of natural numbers and returns a set of natural numbers. All functions in Isabelle/HOL are curried.
`(f a b)`	Function application. Applies the function f on the two arguments. The type of the function is $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$.
`primrec`	Isabelle/HOL keyword used for defining primitive recursive functions.
`constdef`	Isabelle/HOL keyword used for defining non-recursive functions.
`record`	Isabelle/HOL keyword used for grouping multiple fields (attributes) into a single collection.
`theorem`	Isabelle/HOL keyword for defining a theorem.
`lemma`	Isabelle/HOL keyword for defining a lemma. Lemma and theorems are treated the same way.
$\longrightarrow$	Implication inside HOL code.
$\Longrightarrow$	Meta-level implication. Used in Isabelle/HOL code to separate assumptions from conclusions. In our code, where possible we re-format our lemmas to convert $\longrightarrow$ into meta-level implications $\Longrightarrow$.
$[\![\ ;\ ]\!]$	Isabelle/HOL meta-level notation for conjunction $\wedge$. Used for separating assumptions when there are more than one assumptions. Inside the `HOL` code, we use the traditional $\wedge$ conjunction notation for specifying multiple assumptions.

Continued on next page

Table A.1 – continued from previous page

symbols and operations	description
Operations and constructs used in Formalisation	
getName	Field access operation. Returns the Name (identifier) of the component.
getItfs	Field access operation. Returns the component interfaces itf(s).
getQueue	Returns the message queue of a component.
getComputedResults	Returns the results previously computed by a component.
cpList	Returns a list of all subcomponents of a component recursively. cpListList is a mutually recursive auxiliary function of cpList and operates on a list of components.
cpSet	Gives a set representation of elements in the cpList of a component.
cpListSet	Gives a set representation of elements in the cpListList of a component list.
getCp	Retrieves a component from a component lists based on the component name.
^	Shortcut notation for getCp.
getSubCp	Retrieves a component from the list of subcomponents of a component (does not step inside the component hierarchy).
getRecSubCp	Retrieves a subcomponent recursively from all the subcomponents of that component. It searches inside the component hierarchy using cpList. getRecSubCpList is the auxiliary mutually recursive function for component lists.
^^	Shortcut notation for getRecSubCp.
changeCp	Changes a component inside a component list based on component name.
<-	Shortcut notation for changeCp.
removeSubCp	Removes a subcomponent from given component based on subcomponent name.
ComputedRqs	Retrieves the list of request id's for already computed requests in all subcomponents inside a given component. Has an auxiliary function ComputedRqsList.
RegisteredFuture	Verifies if the given future is registered in a component system.

Continued on next page

Table A.1 – continued from previous page

symbols and operations	description
`LocalReferencedRqs`	Gives all the futures referenced from a component, without entering its subcomponents.
`LocalRegisteredFutureComp`	Verifies if all the futures in a given component are registered (without stepping inside the subcomponents).
`GlobalRegisteredFuturesComp`	Verifies if all the futures in a given component system are registered.
`Behaviours`	An LTS defining the possible behaviours of a primitive component.
`Tau`	Internal transition
`NewService`	Request service
`Call`	Request emission
`ReceiveResult`	End of service
`CorrectComponentStructure`	Verifies if the component is well-formed. The auxiliary functions `CorrectComponentStructureList` operates on lists of components.
`CorrectComponent`	Verifies if the component is well-formed and has the correct structure.
`CorrectComponentWeak`	Weak correctness condition. Removes the condition of checking correct future referencing in the hierarchy.
`CorrectComponentWeakList`	Gives the weak correctness condition for list of components.
`CorrectComponentWeakList`	Gives the weak correctness condition for list of components.
`Complete`	Verifies the completeness property for our components. All internal interfaces of a component and all interfaces of its subcomponents should be bound.
Component configuration-reconfiguration	
`unbind`	Removes a binding from the set of bindings inside the composite component.
`Replace`	Replaces a subcomponent while maintaining the complete component property.
`RenameBinding`	Allows to change the `src` or the `destination` of a binding .
Chapter-6	
$[a_i]^{i\in 1..n}$	List of n elements .
$\{a_i\}^{i\in 1..n}$	Finite set of n elements .

Continued on next page

Table A.1 – continued from previous page

symbols and operations	**description**
(a, b)	Pairs.
#	List append operation.
$\backslash$	Remove element from list. $[a_i]^{i \in 1..n} \backslash$ b removes b from list $[a_i]^{i \in 1..n}$.
$[a_i \mapsto b_i]^{i \in 1..n}$	A mapping from a_i to b_i.
$([a_i \mapsto b_i]^{i \in 1..n})[c \mapsto d]$	Add $[c \mapsto d]$ to an existing mapping.
$([a_i \mapsto b_i]^{i \in 1..n})[c \mapsto \emptyset]$	Remove an entry from mapping.
FRL	List of future recipients:$[f_i \mapsto {N_j}^{j \in 1..n}]^{i \in 1..n}$
queue	Message queue of the component: $[R_i]^{i \in 1..n}$
R	Request: $R : (f, v, itf)$ <future, value, interface>
currReq	list of current requests: $[f]^{i \in 1..n}$
refF	list of futures referenced by internal state of component $[f]^{i \in 1..n}$
Enqueue(C, R)	Returns the component C, where the request R is enqueued.
RL	List of future registrations:$[f_i \mapsto {N_j}^{j \in 1..n}]^{i \in 1..n}$
^	`getSubCp` operation.
^^	`getRecSubCp`.
$\leftarrow$	List replacement operator.
RqIdSet(S)	Set of all futures reference in the composite S (`ComputedRqs`). S is a composite representing the component system.
RefFutSet(S)	Set of all futures referenced in component system S.
host(f)	Name of the component computing the result for a future.
cpSet	Set formed of C and all subcomponents of C recursively.
removeResult(f, C, N)	Remove a computed result from inside a component hierarchy.
updateFV(v, f, v')	Update the value of future by replacing every occurrence of the future by the new value.
getName	Returns the name of the given component.
registerListFutures(S, RL)	Registers all the entries in RL in the component system S.

Appendix B

Semantics of Lazy message-based Strategy

Tau

$$\frac{(\mathit{PintState}(s),\, \mathit{Tau}, s_2) \in \mathit{behaviour}(s)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2|\!)], [\,]}$$

Call

$$\frac{(\mathit{PintState}(s), \mathit{Call}(i_1, v, f), s_2) \in \mathit{behaviour}(s) \qquad f \notin \mathit{RefFutSet}(S)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \dashv i_1, f, v \mapsto_O \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2|\!)]}$$

EndService

$$\frac{(\mathit{PintState}(s), \mathit{EndService}(f, v), s_2) \in \mathit{behaviour}(s)}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2, \mathit{results} := \mathit{results}(s) \# [f, v]|\!)], [\,]}$$

ServeNext

$$\frac{(\mathit{PintState}(s), \mathit{NewService}(v, f), s_2) \in \mathit{behaviour}(s) \qquad \mathit{queue}(s) = [f, v, i] \# Q}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s(\!|\mathit{PintState} := s_2, \mathit{queue} := Q|\!)], [\,]}$$

RcvResultPrim

$$\frac{\begin{array}{c}(s, \mathit{ReceiveResult}(f, v), s') \in \mathit{behaviour}(s)\\ s'' = s'(\!|\ \mathit{results} = [f_i \mapsto v_i \mid\ f_i \in \mathrm{dom}(\mathit{results}(s)) \wedge \\ v_1 = \mathit{updateFV}((\mathit{results}(s))(f_i), f, v)],\\ \mathit{queue} = [[f_i,\ v_i,\ \mathit{itf}_i] \mid [f_i,\ v_i',\ \mathit{itf}_i] \in \mathit{queue}(s)\ \wedge\ v_i = \mathit{updateFV}(v_i',\ f,\ v)]|\!)\end{array}}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \dashv f, v, N \mapsto_F \mathrm{Prim}[N, \mathit{itfs}, s'']}$$

WaitByNecessity

$$\frac{\mathit{fidSet} = \mathit{ApplyWait}(\mathit{PintState}(s)) \qquad \mathit{fidSet} \neq \emptyset}{S \vdash \mathrm{Prim}[N, \mathit{itfs}, s] \rightarrow_R \mathrm{Prim}[N, \mathit{itfs}, s], [(f'',\ N) \mid f'' \in \mathit{fidSet}]}$$

Figure B.1: Primitive Component Semantics (Lazy message-based)

Comments The semantics of primitive components with Lazy message-based strategy appear in Figure B.1. The two strategies are closely related, only differing in the time of future registration. Consequently, the two semantics are close to one another. However, in lazy message-based strategy, no registrations are performed when receiving futures or transmitting futures. The registrations are only

performed on *wait-by-necessity*, which triggers future registration (by adding the awaited futures to the RL list).

HIERARCHY
$$\frac{(subCp \uparrow N) \ = \ C \qquad S \vdash \ C \rightarrow_R \ C', \ RL}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[N_0, itfs, (subCp \leftarrow C'), bindings, s], \ RL}$$

COMPOSITECALL
$$\frac{\begin{array}{c} queue(s) \ = \ [f, v, itf] \ \# \ Q \\ f' \notin RqIdSet(S) \qquad s' \ = \ s(\!|queue := Q, \ results := results(s)[f \mapsto (V_f, \{f'\})]|\!) \end{array}}{S \vdash \text{Comp}[N, itfs, subCp, bindings, s] \dashv itf, f', v \mapsto_O \text{Comp}[N, itfs, subCp, bindings, s']}$$

COMMBROTHERS
$$\frac{\begin{array}{c} C = (subCp \uparrow N) \\ [N.itf, \ N'.itf'] \in bindings \qquad S \vdash \ C \dashv itf, f, v \mapsto_O C' \qquad host(f) = N' \\ subCp' = subCp \leftarrow C' \qquad subCp'' = subCp' \leftarrow (Enqueue(subCp' \uparrow N', [f, v, itf'])) \end{array}}{\begin{array}{l} S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \\ \qquad \text{Comp}[N_0, itfs, SubCp'', bindings, s], [\ ] \end{array}}$$

COMMCHILD
$$\frac{\begin{array}{c} queue(s) = [f, v, itf] \, \# Q \qquad [This.itf, N'.itf'] \in bindings \qquad f' \notin RefFutSet(S) \\ host(f') = N' \qquad subCp' = subCp \leftarrow \ (Enqueue((subCp \uparrow N'), \ [f', \ v, \ itf'])) \\ s' := s(\!|queue := Q, \ results := results(s)[f \mapsto (V_f, f')]|\!) \end{array}}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R \text{Comp}[N_0, itfs, subCp', bindings, s'], [\ ]}$$

COMMPARENT
$$\frac{\begin{array}{c} (subCp \ \uparrow \ N) = C \qquad [N.itf', \ This.itf] \in bindings \\ subCp' = subCp \leftarrow C' \qquad S \vdash \ C \dashv itf', f, v \mapsto_O C' \qquad host(f) = N_0 \end{array}}{\begin{array}{l} S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \rightarrow_R Enqueue(\text{Comp}[N_0, itfs, subCp', bindings, s], \\ \qquad\qquad [f, v, itf]), \ [\ ] \end{array}}$$

RCVRESULTCOMPOSITE(1)
$$\frac{\begin{array}{c} s' = s(\!|results = [f_i \mapsto v_i \mid \exists v_i'. [f_i \mapsto v_i'] \in results(s) \ \wedge \ v_i = updateFV(v_i', \ f, \ v)], \\ queue = [[f_j, \ v_j, \ itf_j] \mid \exists v_j'. [f_j, \ v_j', \ itf_j] \in queue(s) \ \wedge \ v_j = updateFV(v_j', \ f, \ v)] \end{array}}{\begin{array}{l} S \vdash \text{Comp}[N, itfs, subCp, bindings, s] \dashv f, v, N \mapsto_F \\ \qquad \text{Comp}[N, itfs, subCp, bindings, s'] \end{array}}$$

RCVRESULTCOMPOSITE(2)
$$\frac{N_0 \neq N' \qquad (subCp \ \uparrow \ N) \dashv f, v, N' \mapsto_F C' \qquad subCp' = subCp \leftarrow C'}{S \vdash \text{Comp}[N_0, itfs, subCp, bindings, s] \dashv f, v, N' \mapsto_F \text{Comp}[N_0, itfs, subCp', bindings, s]}$$

Figure B.2: Semantics of the component composition (a)

Comments The semantics of composite components with lazy message-based strategy appear in Figure B.2. Again, the registration lists are only used with the $\rightarrow_R$ reduction, which may require registrations due to *wait-by-necessity*.

$$\frac{\begin{array}{c}\textsc{TriggerFutureUpdate}\\ C \in \mathit{cpSet}(S_o) \qquad \text{the state of } C \text{ is } s \qquad \mathit{results}(s)(f) = v \qquad \mathit{FRL}(s)(f) = \{N_i\}^{i\in 1..n} \\ \forall i \in 1..n,\ S_{i-1} \dashv f, v, N_i \mapsto_F S_i \qquad S' = \mathit{RemoveResult}(f,\ S_n,\ \mathit{getName}(C))\end{array}}{\vdash S_o \rightsquigarrow S'}$$

$$\frac{\begin{array}{c}\textsc{Global-reduction}\\ S \vdash S \rightarrow_R S', RL\end{array}}{S \rightsquigarrow \mathit{registerListFutures}(S', RL)}$$

Figure B.3: Semantics of the components

Comments Finally, the TriggerFutureUpdate and Global-reduction for our components with lazy message-based strategy are given in Figure B.3. As compared to the eager message-based strategy, the TriggerFutureUpdate rule is simpler, and does not require any registrations. Only the $\rightarrow_R$ reduction may trigger future registration.

Bibliography

[1] Henrio, L., Kammüller, F., Rivera, M.: An asynchronous distributed component model and its semantics. (2009) 159–179

[2] Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)

[3] Halstead, Jr., R.H.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems (TOPLAS) **7** (1985)

[4] Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming abcl/1. In: OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications, New York, NY, USA, ACM (1986) 258–268

[5] Niehren, J., Schwinghammer, J., Smolka, G.: A concurrent lambda calculus with futures. Theoretical Computer Science **364** (2006) 338–356

[6] Johnsen, E.B., Owe, O., Yu, I.C.: Creol: a type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. **365** (2006) 23–66

[7] Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer-Verlag (2005)

[8] Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming. In: OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2005) 31–40

[9] Khan, M.U., Henrio, L.: First class futures: a study of update strategies. Research Report RR-7113, INRIA (2009)

[10] Henrio, L., Khan, M.U., Ranaldo, N., Zimeo, E. Coregrid. In: First Class Futures: Specification and implementation of Update Strategies. Springer (2010) Accepted for publication.

[11] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., Pérez, C.: Gcm: a grid extension to fractal for autonomous distributed components. Annales des Télécommunications **64** (2009)

[12] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer-Verlag (2002)

[13] Henrio, L., Kammüller, F., Khan, M.U.: A framework for reasoning on component composition. In: FMCO 2009. Lecture Notes in Computer Science, Springer (2010)

[14] Henrio, L., Khan, M.U.: Asynchronous components with futures: Semantics and proofs in isabelle/hol. In: Proceedings of the Seventh International Workshop, FESCA 2010, ENTCS (2010)

[15] ProActive: Parallel, Distributed, Multi-Core Computing for Enterprise Grids and Clouds (2008) http://proactive.inria.fr/.

[16] Ranaldo, N., Zimeo, E.: Analysis of different future objects update strategies in proactive. In: IPDPS 2007: Parallel and Distributed Processing Symposium, IEEE International. (2007) 23–66

[17] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions. Springer-Verlag (2004)

[18] Hibbard, P.: Parallel processing facilities. New Directions in Algorithmic Languages (1976) 1–7

[19] JSR 166: Concurrency utilities. http://download.oracle.com/docs/cd/E17476_01/javase/1.5.0/docs/guide/concurrency/index.html (2004)

[20] Taura, K., Matsuoka, S., Yonezawa, A.: Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In: Proceedings of the DIMACS workshop on Specification of Parallel Algorithms, American Mathematical Society (1994) 275–292

[21] Alice ML: Alice project, programming systems lab, saarland university. http://www.ps.uni-saarland.de/alice (2007)

[22] Dedecker, J., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: Ambient-oriented programming in ambienttalk. In: Proceedings of 20th European Conference on Object-oriented Programming (ECOOP), Springer (2006)

[23] Andrews, G.R.: Concurrent programming: principles and practice. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1991)

[24] Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. In: SEFM '04: Proceedings of the Software Engineering and Formal Methods, Washington, DC, USA, IEEE Computer Society (2004) 188–197

[25] Abelson, H., Dybvig, R.K., Haynes, C.T., Rozas, G.J., Adams, IV, N.I., Friedman, D.P., Kohlbecker, E., Steele, Jr., G.L., Bartley, D.H., Halstead, R., Oxley, D., Sussman, G.J., Brooks, G., Hanson, C., Pitman, K.M., Wand, M.: Revised report on the algorithmic language scheme. SIGPLAN Lisp Pointers **IV** (1991) 1–55

[26] McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM **3** (1960) 184–195

[27] Katz, M., Weise, D.: Continuing into the future: on the interaction of futures and first-class continuations. In: LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming, New York, NY, USA, ACM (1990) 176–184

[28] Smith, B.C.: Reflection and semantics in lisp. In: POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM (1984) 23–35

[29] Flanagan, C., Felleisen, M.: The semantics of future and an application. Journal of Functional Programming **9** (1999) 1–31

[30] Flanagan, C., Felleisen, M.: The semantics of future and its use in program optimization. In: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1995) 209–220

[31] Feeley, M.: An efficient and general implementation of futures on large scale shared-memory multiprocessors. PhD thesis, Waltham, MA, USA (1993)

[32] Moreau, L.: The semantics of future in the presence of first-class continuations and side-effects. Technical report (1995)

[33] Hewitt, Baker: Laws for parallel communicating processes. In: Proc. IFIP-77, Toronto (1977)

[34] Rossberg, A., Botlan, D.L., Tack, G., Brunklaus, T., Smolka, G. In: Alice Through the Looking Glass. Volume 5 of Trends in Functional Programming. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany (2006) 79–96

[35] Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)

[36] Smolka, G.: The oz programming model. In: JELIA '96: Proceedings of the European Workshop on Logics in Artificial Intelligence, London, UK, Springer-Verlag (1996) 251

[37] Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, ACM (1988) 260–267

[38] Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, Second Edition: The Java Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)

[39] Welc, A., Jagannathan, S., Hosking, A.: Safe futures for java. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2005) 439–453

[40] Pratikakis, P., Spacco, J., Hicks, M.: Transparent proxies for java futures. SIGPLAN Not. **39** (2004) 206–223

[41] Zhang, L., Krintz, C., Nagpurkar, P.: Language and virtual machine support for efficient fine-grained futures in java. In: PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, Washington, DC, USA, IEEE Computer Society (2007) 130–139

[42] Zhang, L., Krintz, C., Nagpurkar, P.: Supporting exception handling for futures in java. In: PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java, New York, NY, USA, ACM (2007) 175–184

[43] Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17** (1974) 549–557

[44] Grabe, I., Steffen, M., Torjusen, A.B.: Executable interface specifications for testing asynchronous creol components. Technical Report Research Report No. 375, University Of Oslo (2008)

[45] Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming **78** (2008) 491–518

[46] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285** (2002) 187–243

[47] Boer, F.S.D., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Proc. 16th European Symposium on Programming (ESOP'07). Volume 4421., LNCS (2007) 316–330

[48] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. **23** (2001) 396–450

[49] Caromel, D., Henrio, L., Serpette, B.: Asynchronous and deterministic objects. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2004) 123–134

[50] Abadi, M., Cardelli, L.: A Theory of Objects. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1996)

[51] Henrio, L., Kammüller, F., Sudhof, H.: Aspfun: A functional and distributed object calculus semantics, type-system, and formalization. Research Report 6353, INRIA (2007)

[52] Henrio, L., Kammüller, F.: Functional active objects: Typing and formalisation. In: Proceedings of the International Workshop on the Foundations of Coordination Languages and Software Architecture (FOCLASA), Elsevier (2009)

[53] AmbientTalk: Ambient oriented programming. http://soft.vub.ac.be/amop/ (2010)

[54] Miller, M.S., Tribble, E.D., Shapiro, J., Laboratories, H.P.: Concurrency among strangers: Programming in e as plan coordination. In: In Trustworthy Global Computing, International Symposium, TGC 2005, Springer (2005) 195–229

[55] Sessions, R.: COM and DCOM: Microsoft's vision for distributed objects. John Wiley & Sons, Inc., New York, NY, USA (1998)

[56] Grimes, R., Grimes, D.R.: Professional Dcom Programming. Wrox Press Ltd., Birmingham, UK, UK (1997)

[57] Burke, B., Monson-Haefel, R.: Enterprise JavaBeans 3.0 (5th Edition). O'Reilly Media, Inc. (2006)

[58] OMG: Corba component model (v.3.0). http://www.omg.org/technology/documents/formal/components.htm (2005)

[59] OMG: Object Management Group (2010) http://www.omg.org.

[60] Denis, A., Pérez, C., Priol, T., Ribes, A.: Bringing high performance to the corba component model. In: SIAM Conference on Parallel Processing for Scientific Computing. (2004)

[61] Pérez, C., Priol, T., Ribes, A.: A parallel corba component model for numerical code coupling. In: GRID '02: Proceedings of the Third International Workshop on Grid Computing, London, UK, Springer-Verlag (2002) 88–99

[62] Denis, A., Pérez, C., Priol, T., Ribes, A.: Padico: A component-based software infrastructure for grid computing. In: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, IEEE Computer Society (2003) 2.1

[63] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B.: Toward a common component architecture for high-performance scientific computing. In: HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, IEEE Computer Society (1999) 13

[64] Gannon, D., Bramley, R., Fox, G., Smallen, S., Rossi, A., Ananthakrishnan, R., Bertrand, F., Chiu, K., Farrellee, M., Govindaraju, M., Krishnan, S., Ramakrishnan, L., Simmhan, Y., Slominski, A., Ma, Y., Olariu, C., Rey-Cenvaz, N.: Programming the grid: Distributed software components, p2p and grid web services for scientific applications. Cluster Computing **5** (2002) 325–336

[65] CCA-Forum: The Common Component Architecture (CCA) Forum home page (2005) http://www.cca-forum.org/.

[66] Malawski, M., Gubala, T., Kasztelnik, M., Bartynski, T., Bubak, M., Baude, F., Henrio, L.: High-level scripting approach for building component-based applications on the grid. In: CoreGRID Workshop on Grid Programming Model Grid and P2P Systems Architecture Grid Systems, Tools and Environments, Heraklion, Crete, Springer (2007)

[67] Bramley, R., Chiu, K., Diwan, S., Gannon, D., Govindaraju, M., Mukhi, N., Temko, B., Yechuri, M.: A component based services architecture for building distributed applications. In: HPDC '00: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, IEEE Computer Society (2000) 51

[68] Globus: Globus Toolkit (2010) http://www.globus.org/toolkit/.

[69] Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O.: SCA service component architecture, assembly model specification. Technical report (2007) `www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[70] Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)

[71] Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal Component Model. Technical report, ObjectWeb Consortium (2004) `http://fractal.objectweb.org/specification/index.html`.

[72] OW2.Consortium: FraSCAti, Open SCA middleware platform. `https://wiki.objectweb.org/frascati/Wiki.jsp?page=FraSCAti` (2009)

[73] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable sca applications with the frascati platform. In: SCC '09: Proceedings of the 2009 IEEE International Conference on Services Computing, Washington, DC, USA, IEEE Computer Society (2009) 268–275

[74] SENSORIA: Software engineering for service-oriented overlay computers (2005)

[75] Bruni, R., Lafunete, A.L., Montanari, U., Tuosto, E.: Service oriented architectural design. In Barthe, G., Fournet, C., eds.: TGG 2007. Volume 4912 of LNCS., Springer (2008) 186–203

[76] Plasil, F., Balek, D., Janecek, R.: Sofa/dcup: Architecture for component trading and dynamic updating, IEEE CS Press (1998) 43–52

[77] Bures, T., Hnetynka, P., Plasil, F.: Sofa 2.0: Balancing advanced features in a hierarchical component model. In: SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications, Washington, DC, USA, IEEE Computer Society (2006) 40–48

[78] Hnetynka, P., Plasil, F.: Dynamic reconfiguration and access to services in hierarchical component models. In: Proceedings of CBSE 2006, Vasteras, Sweden, LNCS 4063, Springer-Verlag (2006) 352–359

[79] Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In: ICSE '93: Selected papers from the Workshop on Studies of Software Design, London, UK, Springer-Verlag (1996) 17–32

[80] Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28** (2002) 1056–1076

[81] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. Exper. **36** (2006) 1257–1284

[82] Merle, P., Stefani, J.B.: A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA (2008)

[83] Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11** (2002) 256–290

[84] Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed fractal components. Annales des Télécommunications **64** (2009) 25–43

[85] OW2.Consortium: Julia, Fractal reference specification. `http://fractal.ow2.org/julia/index.html` (2010)

[86] OW2.Consortium: Dream, a Component-based communication framework. `http://dream.ow2.org/` (2008)

[87] Baude, F., Caromel, D., Henrio, L., Naoumenko, P. Coregrid. In: A Flexible Model And Implementation Of Component Controllers. Springer (2008)

[88] Cansado, A., Madelaine, E.: Specification and verification for grid Component-Based applications: From models to tools. In: Formal Methods for Components and Objects. (2009) 180–203

[89] Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed fractal components. Annales des Télécommunications **64** (2009) 25–43

[90] Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on grids and P2P systems. Computational Methods in Science and Technology **12** (2006) 69–77

[91] Cansado, A., Canal, C., Salaün, G., Cubo, J.: A formal framework for structural reconfiguration of components under behavioural adaptation. Electron. Notes Theor. Comput. Sci. **263** (2010) 95–110

[92] Tretola, G., Zimeo, E.: Activity pre-scheduling for run-time optimisation of grid workflows. Journal of Systems Architecture **54** (2008)

[93] Tretola, G., Zimeo, E.: Extending semantics of web services to support asynchronous invocation and continuation. In: Proceedings of the IEEE International Conference on Web Services (ICWS). (2007) 208–215

[94] Paulson, L.C.: Isabelle: a Generic Theorem Prover. Number 828 in Lecture Notes in Computer Science. Springer – Berlin (1994)

[95] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle's logics: Hol (2008)

[96] Henrio, L., Rivera, M.: Stopping safely hierarchical distributed components. In: Proceedings of the Workshop on Component-Based High Performance Computing (CBHPC'08) in conjunction with ACM SIGPLAN CompArch 2008. (2008)

[97] Tejedor, E., Badia, R.M., Naoumenko, P., Rivera, M., Dalmasso, C.: Orchestrating a safe functional suspension of gcm components. In: CoreGRID Integration Workshop. Integrated Research in Grid Computing. (2008)

[98] Bannour, B., Henrio, L., Rivera, M.: A reconfiguration framework for distributed components. In: SINTER Workshop on Software Integration and Evolution @ Runtime, ACM (2009)

A Study of First Class Futures: Specification, Formalisation and Mechanised Reasoning

Futures enable an efficient and easy to use programming paradigm for distributed applications. A future is a placeholder for result of concurrent execution. Futures can be *first class* objects; first class futures may be safely transmitted between the communicating processes. Consequently, futures spread everywhere. When the result of a concurrent execution is available, it is communicated to all processes which received the future. In this thesis, we study the mechanisms for transmitting the results of first class futures; the *future update strategies*. We provide a detailed semi-formal specification of three main future update strategies adapted from ASP-calculus; we then use this specification for a real implementation. We study the efficiency of the three update strategies through experiments. Ensuring correctness of distributed protocols, like future update strategies is a challenging task. To show that our specification is correct, we formalise it together with a component model. Components abstract away the program structure and the details of the business logic; this paradigm thus facilitates reasoning on the protocol. We formalise in Isabelle/HOL, a component model comprising notions of hierarchical components, asynchronous communications, and futures. We present the basic constructs and corresponding lemmas related to structure of components, and formal operational semantics of our components in presence of a future update strategy; proofs showing correctness of future updates are presented. Our work can be considered as a formalisation of ProActive/GCM, and shows the correctness of the middleware implementation.

Une Étude des Futurs de Première Classe: Spécification, Formalisation et Preuves Formelles.

Les futurs fournissent une modèle de programmation efficace pour le développement des application distribuées. Un futur est une objet temporaire qui représente le résultat d'une exécution concurrente. Les futurs peuvent être des objet de «première classe», et ainsi être transmis en toute sécurité entre les processus communicants. En conséquence, les futures se propagent partout dans le système. Lorsque le résultat d'une exécution simultanée est disponible, il est communiquée à tous les processus qui ont reçu le futur. Nous étudions les mécanismes de transmission des résultats des futurs; les "stratégies pour mise à jour des futurs". Nous fournissons une spécifications semi-formelle détaillées, de trois principales stratégies. Nous utilisons ensuite cette spécification pour une véritable implementation et nous étudions l'efficacité des trois strategies. C'est une tâche difficile d'assurer la correction des protocoles distributes. Pour montrer que notre spécification est correcte, nous la formalisons avec un modèle de composants. Les composants abstraient la structure du programme; ce paradigme facilite donc le raisonnement sur le protocole. Nous formalisons dans Isabelle/HOL un modèle de composants comprenant des notions de composants hiérarchiques, les communications asynchrones, et les futurs. Nous présentons les constructions de base et des lemmes liés à la structure des composants. Nous présentons une sémantique formelle des nos composants en présence d'une stratégie de mise à jour de futurs; Les preuves montrant la correction de notre stratégie sont présentées. Notre travail peut être considéré comme une formalisation de ProActive / GCM.