UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

etsinf

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Proyecto Fin de Carrera / Projecte Fi de Carrera

Para optar a la titulación de / per a optar a la titulació de

Ingeniero de Informática

presentado por / presentat per

Nikolas Martens

Dirigido / tutorizado por
dirigit / tutorizat per

Javier Jaén

Valencia, Febrero 2011

Design and Implementation of a Distributed Software Platform Based on Asynchronous Messages

Nikolas Martens
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Advisor: Javier Jaén

Diploma Thesis

February 2011

Acknowledgements

I would like to thank my advisor, Javier Jaén, for his valuable feedback and support in many ways during the projects and Prof. Klaus Diepold who gave me the opportunity to write this thesis. I also feel deepest gratitude to my friend Robert Jenke who gave me the idea and the courage to write about this topic. Lastly, I thank my family for their unconditional support.

Abstract

Although personal computing has advanced and spread considerably during the last decades, the real computer revolution has not happened yet. In order to promote computer literacy, a system is needed that enables the user to use the power of abstraction for building and arguing about dynamic models. This thesis presents a programming model that follows the paradigm "everything is an object" [1] to the utmost in order to build a software platform upon it that provides this possibility. The result is a lean yet powerful model based on a single kind computational units which form a virtual network. All entities of the model are therefore concurrent, distributed and persistent. Like objects, they encapsulate behaviour and state and communicate through message passing, but unlike conventional objects, an entity has no methods, messages are asynchronous, the only structure is composition and the only association is specialization.

Contents

Chapter 1

Introduction

Personal computing has made remarkable advances and became omnipresent during the last decades but like Turing Award winner and personal computing pioneer Alan Kay states repeatedly "the computer revolution has not happened yet" [2]. Computers have not yet had an impact comparable to the printing press for example, which not only changed completely how argumentation was done but also what was argued about. This change took approximately 150 years to happen so it is not a surprise that 30 years after their introduction, personal computers are still used as a faster and cheaper imitation of paper, recordings, films and television.

In order for a computer revolution to happen, *computer literacy* needs to reach a critical mass, which is not only the ability to read and write but also to understand and argue about ideas that are worth writing about. What words are to printing, dynamic models are to computing, thus computer literacy is the ability to analyse and build models, understand the ideas they represent and use them to argue about these ideas.

This requires tools to build and share dynamic models like letters and books are used for printing. But in computer science still no common set of letters exists and the tools are only available to a small group of specialized users called programmers. Two of the most powerful tools are abstraction and distribution.

1.1 Abstraction

The single most important principle of computer science is abstraction [3, 4, 5]. It began with the division of software and hardware in the *von Neumann* architecture and was continued with the appearance of programming languages and operating systems. With the increasing computational power, every generation of programming languages used a higher level of abstraction of the machine it is running on.

Not only the abstraction of the programming models has evolved but also the ability to create new abstractions with the language [6]. This development led from the first machine language over assembly languages to high level languages. At the cost of performance, each generation increased the expressiveness of the programming language and thus decreased the time necessary for development and maintenance. As a result, modern object-oriented languages do not require any knowledge of the underlying hardware and can be used efficiently to model dynamic software systems.

Besides the ability to build more complex systems in less time, abstraction also brings the advantages of re-usability and interoperability [7]. By hiding differences and emphasizing similarities, abstract models can be re-used for different but similar problems and communicate with each other more easily by ignoring unimportant details.

All of these advantages however, are restricted to the production of software. Because internals of a computer program are invisible to the end user, he can not benefit from the concepts used to build it. The world of the end user, which consists of files and programs has little connection with the world of objects, which resides inside programs. This affects software developers more than users since most programs work around this deficiency and hide it from the user. But few programs or software platforms let their users profit from the full power of abstraction. As a consequence, in personal computing there are many almost-the-same things which have to be treated differently because there exists no abstraction for them. An example are common problems with different file formats and encodings for documents, pictures and videos which require different viewers. Also web pages and local programs are nowadays quite similar but work differently for historical reasons.

It is also due to historical reasons that many modern programming languages lack of consistency [8]. Some of the most widespread languages for example use a hybrid approach for backwards compatibility [9] and to allow access to lower abstraction levels [10]. While different levels of abstraction are important to optimize efficiency, the lack of consistency increases the complexity of the programming models and compromises their expressiveness. These approaches also violate the core system building principle of separating meaning and optimization [11]. But even the most purely object-oriented languages involve parts which are not objects or substantially different, such as packages, modules, methods or closures. Other components are not even part of the object model and can not be abstracted at all or only with difficulties such as location, file systems and databases.

1.2 Distribution

When the World Wide Web was designed for sharing scientific documents, its creators did not anticipate the way it was going to be used twenty years later. The strength

and weakness of HTTP and HTML is simplicity and the resulting proliferation. Due to its limitations, a vast ecosystem of complement technologies has emerged to provide rich internet applications without compromising accessibility. Nowadays, even a simple dynamic web page involves at least four different technologies plus the knowledge of differences amongst interpreters.

Despite these hurdles, the popularity of internet applications has increased significantly over the last decade. This development has led to a browser-centric usage in personal computing today which resembles the use of terminal computers in the '80s. Internet applications run mostly on powerful servers with a very slim client which provides little more than a graphical interface for user interaction. For the software producers this brings many advantages in the areas of deployment, maintenance, scaling and not at last control over user data. The user does not only have to accept the lost of control over his data but also has do deal with conflicting software models which results in bookmarks and "back" buttons being mostly useless in internet applications.

A software platform which is aimed at the needs of modern personal computing would provide the the possibility to connect to any other system and share any kind of data in a safe and transparent way. To be able to do so, data must not be static but consist of dynamically responding entities that carry all information with them that is needed for presentation and manipulation. Also, these entities can be communicated with and their abilities explored dynamically. Using a common minimal interface for presentation and interaction, all parts of such a system could be used interchangeably when used to assemble new systems.

The biggest increase of productivity in software development does not come from more powerful tools or programming languages but from software reuse [12]. The majority of programming tasks consists of connecting existing software modules rather than creating new ones. But still, the wheel is re-invented many times since there is no wide-spread completely distributed system of dynamically explorable modules. Such a system would give every user access to a global collection of already existing solutions and also the ability to share his own ideas with the world.

It would also leverage the power of connection to create a *personal internet* whose users are able to connect their own devices as peers to form a virtual computer whose data is distributed over the peers and only dynamically cached on the devices. This way any personal information and programs can be accessed in the same manner as any other resource made available by other users without compromising privacy but also without having to distinguish between local and remote services. There does not even exist a distinction between data and programs since all parts of the system are dynamic.

1.3 Aims

In order to build a completely coherent personal computing system that separates meaning from optimization, it has to be based on a programming model that incorporates the concepts of abstraction and distribution. It has to enable the user to dynamically model software systems inside a virtual space in a manner close to natural processes. This programming model has to be implemented into a runtime environment that supports the complete personal computer platform.

The result is a model of virtual objects which exist independent of any hardware. Computers only serve as portals that connect the virtual with the real world. Objects are living things that can be interacted with by sending them messages. To keep the communication as simple as possible, it is unidirectional, the receiver always reacts in the same way and the message is another object. Systems that are too complex to be modelled by a single object are composed by objects in *has-a* relationships. And to allow abstraction, objects also have *is-a* relationships with other objects.

The model is an abstraction of other object-oriented programming models, based on asynchronous message passing, prototypes and inheritance. It is an abstraction because it uses only one entity to encapsulate behaviour and state which plays the roles of classes, objects, methods and closures. Furthermore, it imposes a minimum of structure and protocol to increase interoperability and extensibility. It also can be arbitrarily nested and uses specialization and inheritance. The model is completely distributed which means it treats all objects equally, regardless of their location. As a consequence, it is also completely concurrent since all computation can possibly be carried out on different systems.

1.4 Characteristics

These requirements are fulfilled by the programming model presented in this thesis which has the following characteristics. To the best of our knowledge, no other software platform or programming language possesses all of these characteristics.

Completely concurrent. Objects are independent of processor units and react instantly to a message. The only operation is therefore asynchronous message passing. The model includes a form of data flow for synchronization.

Completely late bound. Messages can only be sent to an object using its address. An address is always name-based thus all communication is late bound.

Completely distributed. Objects are also independent of physical memory so they are not bound to any device but are distributed in a network connecting an arbitrary number of devices. Name resolution is performed by the objects, leading to a distributed directory.

Completely persistent. Since objects do not depend on any hardware, they are not destroyed if a program is closed or a device shut down. Once created, an object continues existing until deleted explicitly.

Completely dynamic. All objects of the model can have behaviour, i.e. are dynamic and living entities. New objects are created and existing ones modified by sending messages to other objects. Hence there is no intrinsic distinction between compile and run time.

Minimal structure. Objects do not have any internal structure but can form compositional hierarchies. Thus objects serve as modules by containing other objects in a part-whole relationship. There are no classes, but objects serve as prototypes for new objects. Objects also do not have methods but always react with the same behaviour.

Minimal association. Except composition, the only association between objects is specialization. An object that specializes another object inherits all of its properties.

1.5 Outline

This thesis presents the implementation of the described programming model as the prototype of a software platform. Methods and techniques used during its development are described in the following chapter.

The architecture of the resulting platform is illustrated in Figure 1.1 with the outline of the chapters describing its parts indicated by arrows. Chapter 3 contains a detailed description of the programming model supported by a biological metaphor, an iterative example and the implementation of the platform's kernel. The kernel's environment is presented in Chapter 4 consisting of mechanisms to connect the kernel to the local and remote systems and library support for reflection and generic components. Chapter 5 describes graphical development tools on application level.

The remaining chapters contain a discussion of results of the project and an outlook on future challenges along with conclusions.

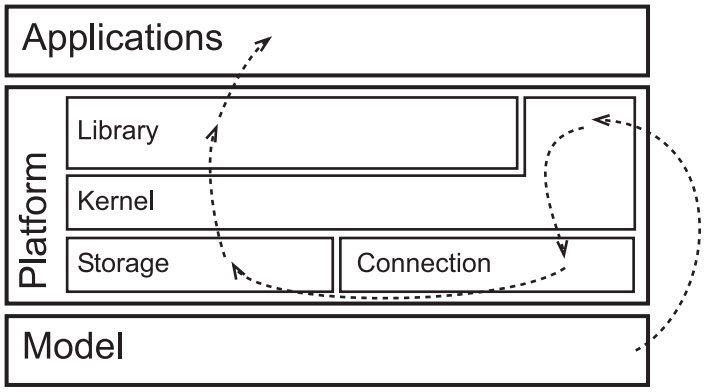


Figure 1.1: Architecture of the software platform and outline of this document.

Chapter 2

Methods

This chapter describes the methods and techniques used during the development of the presented software platform. Due to the explorative nature of the project, methods supporting an iterative style of development were chosen. The following sections describe different approaches to software engineering and two applied methods in more detail.

2.1 Software Development

According to the IEEE, software development is part of software engineering which is defined as (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1). [13]

There are generally two classes of methodologies, independent of their field of application. The *top-down* method starts with the abstract and proceeds to the concrete while the *bottom-up* follows the opposite direction. [14]. In software development, the two classes correspond to the general fields of *structured* methods and *agile* approaches, respectively. While structured methods focus on predictability, agile development puts more emphasis on adaptation.

The NATO Software Engineering Conference in 1968 [15] marks the beginning of systematic approaches to software development in order to address issues caused by the complexity of computer programs [16]. Early manifestations of this approach are the waterfall model and structured programming. As illustrated in Figure 2.1, the former models the development process as a linear sequence of several phases, with the output of each phase being the input of the next. While its strict linearity was

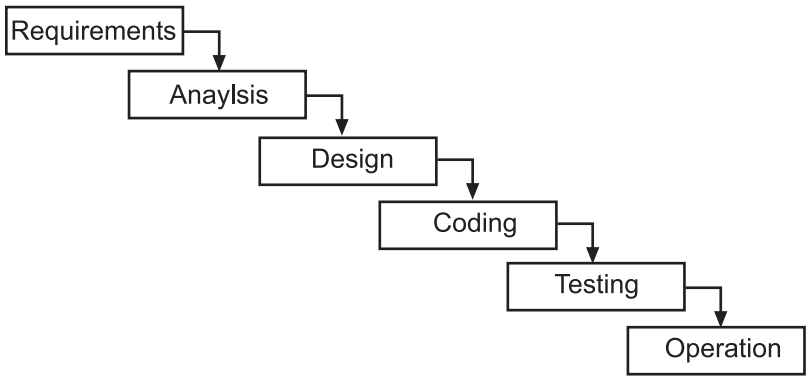


Figure 2.1: A waterfall process model with six phases: Requirements, Analysis, Design, Coding, Testing and Operation.

already described by its introducer as being risky and inviting failure [17], the model has reached widespread acceptance and popularity.

The waterfall process model is an extreme on the continuum between top-down and bottom-up methods and is only applicable for solutions to very well understood problems which can be completely planned and designed in advance. Since in practice this type of project is very rare, modified waterfall models have been introduced which allow iteration by providing feedback channels between certain or all phases.

Agile software development is placed on the other end of the continuum and recommends many iteration of the complete process in very short cycles in the magnitude of weeks. It uses process models such as the spiral model which adds the dimension of *completeness*. Figure 2.2 illustrates the transition from a iterative waterfall model to the spiral model. According to this model, the development runs through all phases with only a small subset of the product's requirements and increases the set with each iteration.

Although used in practice as early as the 1950s, public awareness of agile development methods was very low until the 1990s when most standardized method emerged such as Scrum, Feature Driven Development (FDD) and Extreme Programming (XP) [18]. These methods gained very quickly popularity especially in the development of internet applications. The term *agile development* was introduced in 2001 by the *Manifesto for Agile Software Development* [19].

In practice, software projects use a methodology located between these two extremes, considering project and team size, budget, time limits and other influences. More stable and well understood projects use fewer and longer iteration cycles while more risky and dynamic projects are best suited with shorter cycles and releasing often.

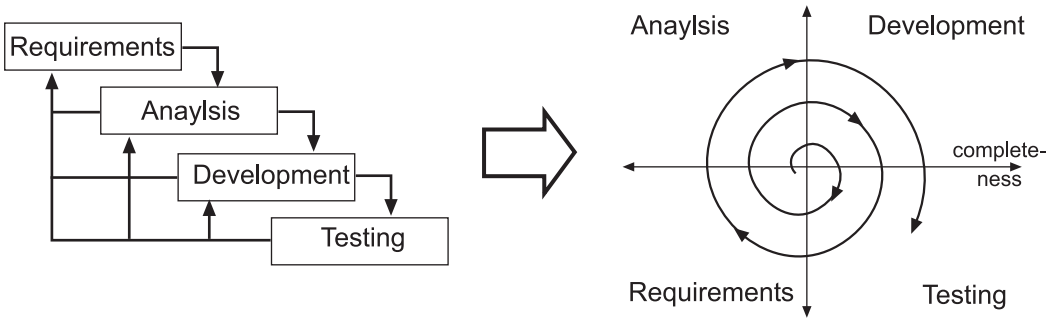


Figure 2.2: Transistion from linear waterfall model to iterative spiral model using feedback channels.

2.2 Prototyping

Due to its explorative nature, an agile software development method was used in this project called *rapid prototyping*. A prototype is a executable program which does not implement all features required for the final product but only those that serve a specific purpose. The method follows the spiral model illustrated in Figure 2.2 where each iteration produces a prototype which is used for experiments to revise existing requirements and determine new ones.

The advantage of prototyping is a tight feedback loop. Errors in analysis and design are identified early while the cost of changes is still low. Prototypes are also a useful mean to facilitate communication with a client or amongst developers since concrete implementations can be discussed instead of abstract descriptions. Disadvantages of prototyping are increased costs during early stages of a project and the risk of distraction from proper analysis which may lead to a decreased robustness. These characteristics also hold true for other iterative development methods. The unique character of prototyping is that it prototypes do not necessarily form the base of future iteration but are often thrown away.

During this project, prototyping was used to verify and improve the programming model. The model was implemented with low cost into a prototype which was then used to verify the model using experiments. Several level of experiments were used reaching from simple message passing in different cell system constellations to complete example applications. Each prototype lead to an improved version of the programming model or the binding algorithm which was implemented into a new prototype. In total, eight iterations were traversed during the project.

2.3 Test-Driven Development

A technique tightly coupled with agile software development is *Test-Driven Development* (TDD). As shown in Figure 2.3, the testing and development phases of the waterfall model are swapped, putting testing first. The model is best suited for the development of small, independent functional units which can be validated by automated tests called *unit tests*. This proceeding has several advantages: [20]

Better tests. Programmers tend to avoid critical constellations – mostly unconsciously – if they exactly know what the tested code is doing. Tests written beforehand are less prejudiced.

Better design. When writing a test first, the developer has to think about a module's interface before its implementation. Writing small, clearly structured and loosely coupled modules is enforced since convoluted systems are not testable. If a module cannot be tested, it is refactored.

Confidence. Being able to test each part of a system at any time, project members become more confident about its reliability. Refactoring becomes less dangerous and is performed more often [21].

Higher test coverage. Writing tests is a tedious task and mostly not done because of time pressure. Doing it first requires less discipline and results in more tests.

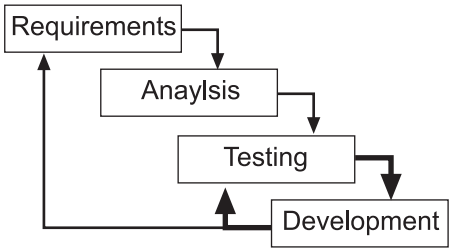


Figure 2.3: Process model of test-driven development.

The applicability of TDD is limited by the range of automated testing. Concurrent processes, graphical user interfaces and system integration can not or only with difficulty be tested automatically [22]. In some cases the method can still be applied by designing tests which require manual validation before implementation.

In this project, TDD was applied with great success to implement all of the described parts of the software platform. The characteristics of automated testing as executable requirements specification proved especially useful with each new prototype. While the

design of the system changed significantly between two iterations, the test suite almost did not changed and thus ensured that the new prototype provides all features of the last one including all bug fixes. In average, each prototype used approximately 70 unit tests.

Test-Driven Development is to a development cycle like the development cycle to the project. It provides a tight feedback loop and enables the developer to solve a problem in small steps. Like development cycles, the size of the steps can be adapted to the problem. Big steps can be taken on well known ground and small steps on new ground or when a big steps fails.

Chapter 3

Programming Model

This chapter describes the programming model in a top-down fashion. The first section describes the concepts and design of the model, followed by the demonstration of these with the help of an example. The last section describes the implementation of the model. It covers only the parts which directly derive from the model, further parts of the kernel are described in Chapter 4.

3.1 Concepts

The programming model is presented in this section by introducing its concepts individually using the metaphor of a biological cell as suggested in [23].

3.1.1 Cells

The model consists of a single kind of behavioural building block called *cell*. Like biological cells, these virtual cells live independent of each other and are able to react concurrently. Cells can be nested arbitrarily. Nested cells are called *children* and the containing cell *parent*. This results in a hierarchical structure with a global *root* cell named "$\circ$". Figure 3.1 illustrates nested cells.

3.1.2 Names and Paths

Each cell has a name which is unique amongst its siblings. A parent cell can address one of its children using its name. Children can address their parent cell using "parent" and the root cell with "$\circ$". A cell can always address itself using "self". Cell names can

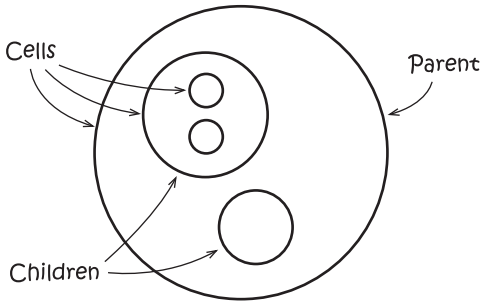


Figure 3.1: A single kind of hierarchically structured behavioural building block.

be combined to form a cell *path* which leads from an origin to a destination cell. Cell paths are analogue to paths in file systems where ".", ".." and "/" correspond with "self", "parent" and "∘" respectively. These references are called *aliases*.

Figure 3.2 illustrate an example of nested cells with names and aliases. In this example, the path Roots.parent.Trunk.Branch (using dots to separate cells) leads from Tree to Branch.

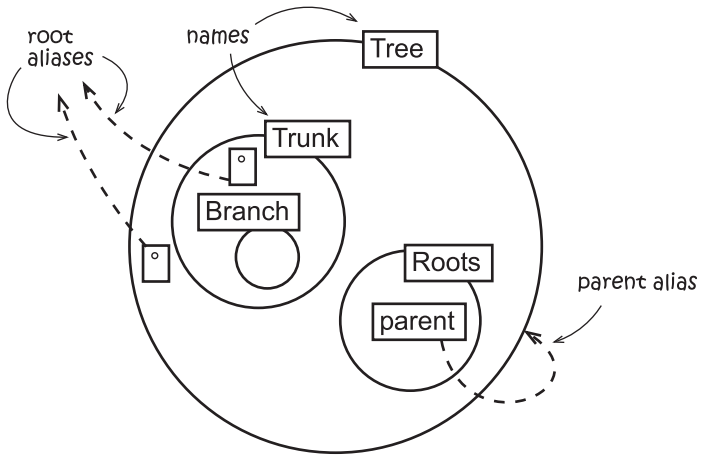


Figure 3.2: Nested cells with names and aliases.

3.1.3 Messages

The only way of communication between two cells is by sending messages. A message is a cell which is sent from a sender to a receiver cell as illustrated in Figure 3.3. Message passing is always asynchronous which means that the sender does not wait for the message to be received and processed.

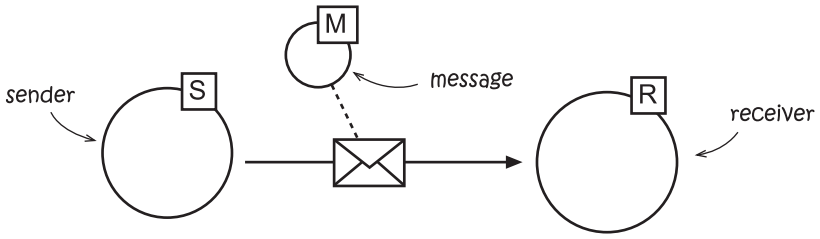


Figure 3.3: A message is being sent from a sender to a receiver cell.

3.1.4 Reaction

A cell reacts on the event of receiving a message. All cells contain a description of how to react called *reaction* which is executed each time the cell receives a message. As shown in Figure 3.4, a reaction consists of *mailings*, each containing the paths of the receiver and message cell, both relative to the sender. During execution, each mailing of the reaction is processed simultaneously, i.e. each message is sent to its corresponding receiver cell.

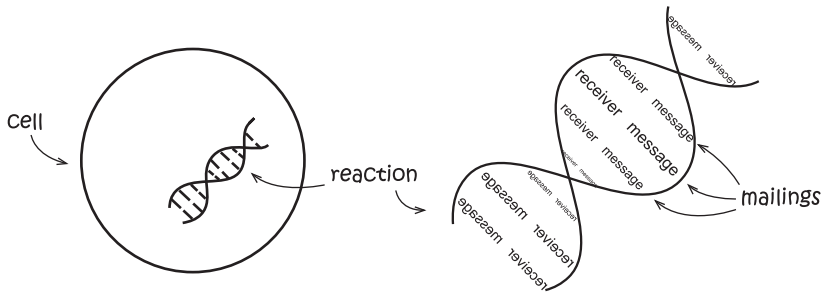


Figure 3.4: A cell with its reaction consisting of mailings.

3.1.5 Specialisation

As with biological cells, the only way to create a new cell is by *specializing* an existing cell. The created *sub-cell* inherits all properties of the *stem* cell which includes children and reaction. As depicted in Figure 3.5, the sub-cell can add further children or replace an inherited child or reaction, but it can not remove an inherited child. The relationship is strictly unidirectional and dynamic. This means that no change of the sub-cell can affect the stem cell but changes to the stem cell affect all sub-cells instantly. Also, the stem cell of an existing cell can be changed.

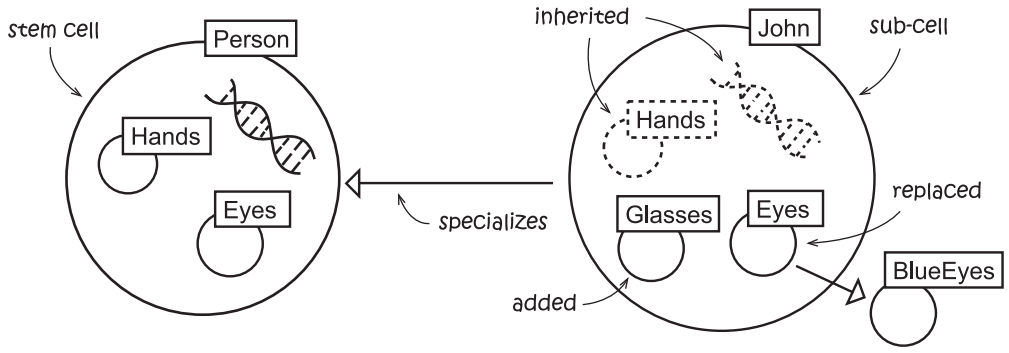


Figure 3.5: John specializes Person, inherits Hands, replaces Eyes and extends it with Glasses.

3.1.6 Execution

The reaction is not executed by the receiver cell directly but by a new child which specializes the receiver. This *execution cell* extends the receiver by the alias message which is connected with the message cell as shown in Figure 3.6. Aliases can be compared with hard links in file systems thus X.message.parent does not resolve to X but to the parent of M. The execution cell can also be extended by further children and therefore provide a storage space for local results which only matter to the execution.

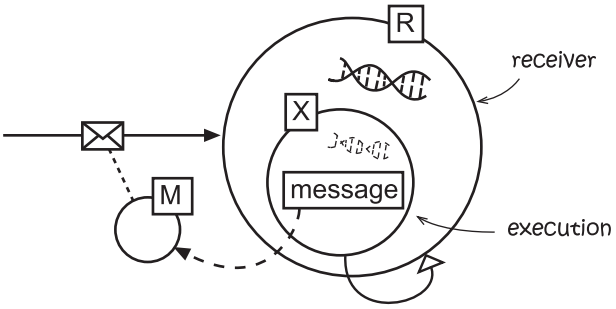


Figure 3.6: A receiver creates a new execution that executes the reaction and which extends the receiver with an alias to the message cell.

3.1.7 Delivery

A message might not be delivered because of three possible reasons: an error, a non-existent receiver or a deactivated receiver. In either case, the message is re-sent until delivered successfully or explicitly cancelled. This enables messages to be sent to cells

before they exist, e.g. results before they are calculated, which can be used for data flow synchronization as illustrated in Figure 3.7. A deactivated cell can not receive any message and deactivates all children except those that give access to its internal properties such as stem, reaction and children. This way cells with incomplete definitions can be made inaccessible.

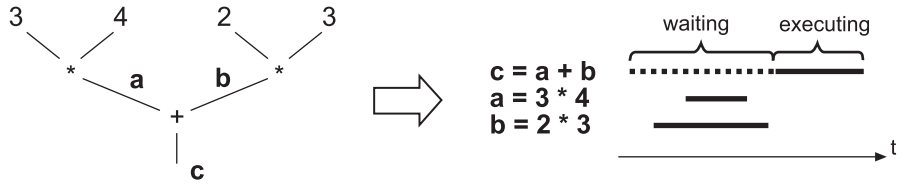


Figure 3.7: Data flow synchronization of the the calculation $3 \cdot 4 + 2 \cdot 3$. The summation is executed earlier but waits for both multiplications.

3.2 Example

The presented concepts of the programming model are illustrated in this section using an example application. The example is divided into two versions. A simple version illustrates the most basic concepts and an extended version illustrates the remaining concepts.

Cells are defined in tables containing the hierarchy, name, stem cell path and reaction of the cell. Paths are written using dots to separate parents and children where parent references are abbreviated with ρ. The mailings of reactions are written as "$receiver \leftarrow message$" where *receiver* and *message* are paths relative to the defined cell. Instead of using mailings, reactions may be described informally or left undefined and therefore inherited by the stem cell.

3.2.1 Simple Version

The example application is a simple publish-subscribe system. Several cells can *subscribe* to a channel and, as a result, receive all messages that have been *published* on that channel.

Definition

Table 3.1 contains the definitions of the involved cells. Top-level cells are children of the root cell.

Table 3.1: Cell definitions of simple publish-subscribe example application.

Cell	Stem	Reaction
Channel	Cell	$subscribers.each \leftarrow forwardMessage$
|_ forwardMessage	Cell	$message \leftarrow \rho.\rho.message$
|_ subscribers	List	
Cell	Cell	Does nothing
List	Cell	
|_ each	Cell	Sends each element to its message
|_ add	Cell	Creates new element with message as stem

The first three rows define cells specific to the publish-subscribe application. The following definitions are generic cells which are part of a standard library and included for completeness. The specific cells are Channel with its two children forwardMessage and subscribers. The cell Cell is the default stem cell and the root of all specialisation hierarchies. Figure 3.8 illustrates the defined cells and their relationships.

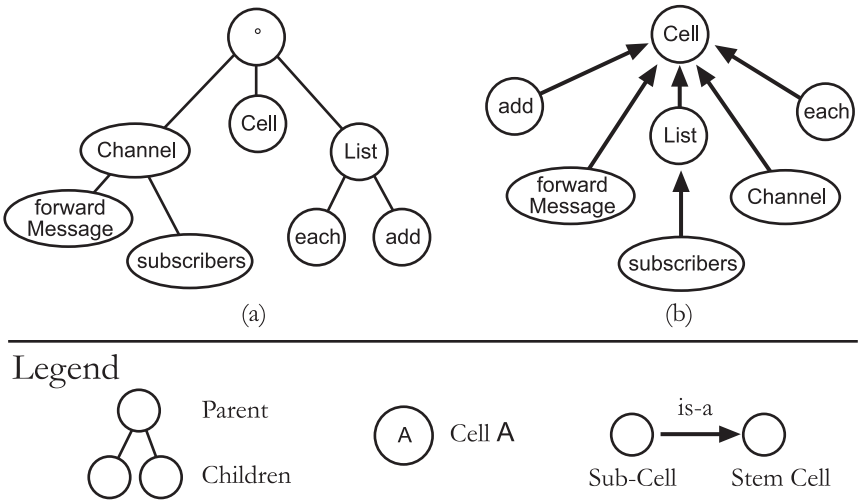


Figure 3.8: Composition (a) and inheritance (b) hierarchies of defined cells for a simple publish-subscribe system.

A cell can be published on the Channel by being sent to it as a message. Every time the Channel receives a message, its reaction is executed, i.e. its child forwardMessage is sent to the child named each of Channel's child subscribers. Note that subscribers inherits each from List. The reaction of each is implemented on the kernel level and sends each of the list's elements to the cell that was received by each, in this case forwardMessage.

As a result, forwardMessage receives each of subscribers elements. The instruction

of forwardMessage's reaction is then executed with each subscriber as its message and sends the published cell (the message received by Channel) to each subscriber (message sent to forwardMessage by subscribers.each).

Execution

The two parent references (abbreviated with ρ) in the reaction of forwardMessage are necessary because the reaction does not execute in the context of the receiver cell but in the context of its execution cell as described in Section 3.1.6. If cells play the role of methods, executions play the role of their activation records. Unlike activation records, executions do not cease to exist when the reaction completes since executions are accessible from within other executions and due to the asynchronism of message passing they do not become unreachable.

The execution cell is a specialisation of the receiver cell, therefore inherits all of its children and can send them messages as done in the reaction of Channel. Thus the first parent of forwardMessage's reaction references the forwardMessage cell and the second parent references the execution of Channel's reaction.

Figure 3.9 illustrates executions and message aliases with a subset of the cells involved in the example application. Executions are depicted as name-less cells which are children and sub-cells of the receiver cell and have a message alias as child. In the example, each sends a message to its message child (see Table 3.1) but because of the alias, the message is received by forwardMessage (labelled "fm" in the figure). Because the reaction of forwardMessage runs within its execution, the channel's message is the message child of its execution which is two parents up from forwardMessage's execution.

Usage

To actually use the system, subscriber cells have to be added to the channel and cells be published. Table 3.2 contains the definitions of subscribers and the driver cells Initialize and Run, whose reactions add subscribers to a channel and publish a literal string on it.

3.2.2 Extended Version

In this section new features are added to the publish-subscribe system of the previous section to illustrate further concepts of the software platform. In the extended version of the application, a subscriber cell can influence whether it receives a published cell or not. This is done by adding subscriptions to a channel which contain logic to decide for each published cell whether its subscriber is interested in it or not. When a new cell is published, the channel sends it first to each subscription and only forwards it to the subscriber if the subscription replies positively.

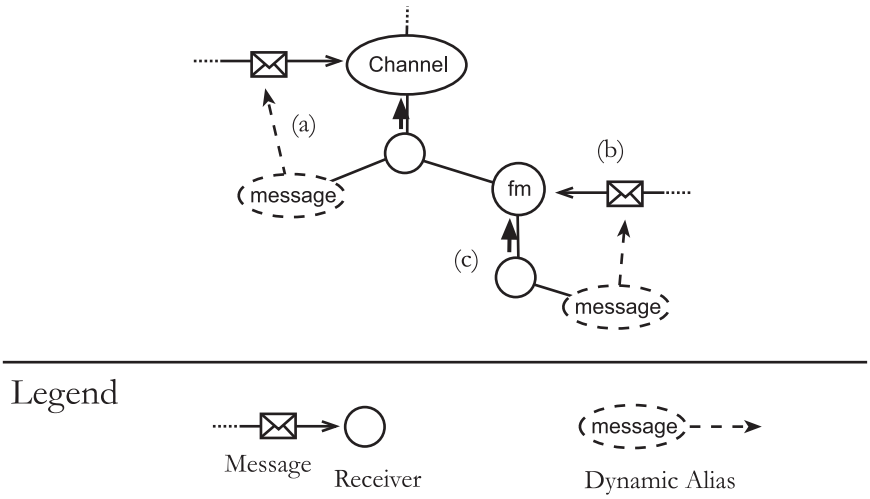


Figure 3.9: Mailings and resulting execution cells of the simple publish-subscribe system. **(a)** A publisher sends a message to Channel which creates an *execution* with a message alias as child. **(b)** Eventually forwardMessage (fm) receives a subscriber as message and creates an execution as well. **(c)** The execution is a child of forwardMessage so the published cell is its parent's parent's message.

Table 3.2: Definition of subscribers and driver cells to run the publish-subscribe example.

Cell	Stem	Reaction
subscriber1	Cell	Does something with its message
subscriber2	Cell	Does something with its message
Initialize	Cell	$\circ.Channel.subscribers.add \leftarrow \circ.subscriber1$
		$\circ.Channel.subscribers.add \leftarrow \circ.subscriber2$
Run	Cell	$\circ.Channel \leftarrow \circ.Literal.String."HelloWorld"$

Responses

Because message passing is asynchronous, a sender that expects a response has to send a cell along with the message that the receiver can respond to. This is done by creating a *container* cell which specializes the actual message. By convention, the cell that the response is expected to be sent to is a child of the container named respond.

Figure 3.10 compares mailings (a) without and (b) with response. In the first case, A sends M to B which cannot send any message as response since it has no information about A. It could send a response to M but if the same cell is sent more than once a correlation between messages and responses would be impossible. For this reason, A sends a unique container cell C (letter I in Figure 3.10 (b)) which specializes M and contains a cell R, to which B sends its response message (letter II).

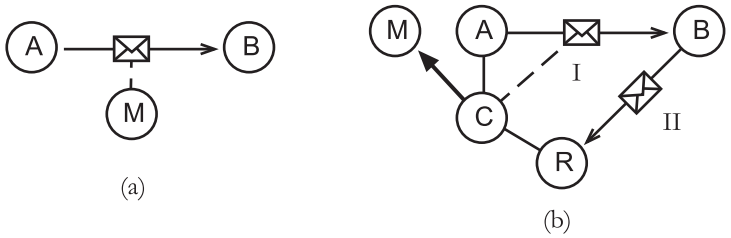


Figure 3.10: Cells involved in mailing (a) without and (b) with response.

Definition

The involved cells are defined in Table 3.3, which also contains the definitions of further library cells needed by the example. These definitions are complementary to previous definitions.

Table 3.3: Cell definitions of extended publish-subscribe example application.

Cell	Stem	Reaction
Channel	Cell	$subscribers.each \leftarrow forwardMessage$
|		$subscriptions.each \leftarrow askSubscription$
|_ subscriptions	List	
|_ askSubscription	Cell	$cell.create.wants \leftarrow \rho.\rho.message$
|		$message \leftarrow wants$
|		$wants.response.ifTrue \leftarrow forwardMessage$
|_ forwardMessage	Cell	$\rho.\rho.message.subscriber \leftarrow \rho.\rho.\rho.message$
Subscription	Cell	$message.respond \leftarrow \circ.True$
Cell	Cell	
|_ respond	Cell	$\rho.\rho.cell.create.response \leftarrow message$
True	Cell	
|_ ifTrue	Cell	$message \leftarrow \circ$
False	Cell	
|_ ifTrue	Cell	does nothing

The first definition extends the reaction of Channel by a second instruction which iterates through the subscriptions list (second definition) by sending the channel's child askSubscription to its child each.

The third definition contains the reaction of askSubscription to which all the subscriptions of the channel are sent individually as messages by each. The first instruction creates a new cell named wants with the published cell (message of the channel) as its stem. The new child is sent to the received subscription in the second instruction.

The new cell is created using cell.create which is implemented on the kernel level and has an infinite number of children that create cells with their own name and the

received message as stem. Note that wants is created as a child of the execution cell and not of askSubscription. In this case, the execution serves as a local name-space just like activation records. This way executions created by further subscriptions sent to askSubscription are able to create their own wants children without conflicting with other executions.

The child wants plays the role of the before mentioned container cell which is necessary because it contains respond (inherited by Cell) which will receive the response. The reaction of respond creates a cell named response in the receiver's parent with the received cell as its stem.

This behaviour is used by a feature of the programming model called *data flow synchronization* which is described in Section 3.1.7. The third instruction of askSubscription's reaction sends a message to a child of wants.response. This cell does not exist before wants.respond has received a message. The instruction can still be executed because in the case of a non-existent receiver, a mailing is repeated until its message is delivered. This enables instructions to be processed before all of their required information is available.

The same instruction is also an example of library-based control structures. The reaction of forwardMessage is only executed if the response is a sub-cell of True since only True defines a reaction for its ifTrue child.

The last definition for the system is a prototype of Subscription which provides a default reaction by responding with ∘.True.

Usage

As in the previous section, driver cells are needed in order to run the application. The cells defined in Table 3.4 create two channels by specializing Channel and two subscriptions for subscriber1 of the previous example. The first subscription inherits the default reaction from its stem cell Subscription and the second subscription executes its stem cell's reaction explicitly by forwarding its message to its child stem which works like *super* in conventional object-oriented languages. And as before, cells are defined to initialize and run the example.

3.3 Implementation

This section describes the implementation of the object model, how mailings are processed, cell paths bound to executions and other functional parts that derive directly from the programming model. Parts of the implementation regarding the environment of the software platform such as storage, distribution, reflection and libraries are described in the next chapter.

Table 3.4: Definition of channels, subscribers and driver cells to run the extended publish-subscribe example.

Cell	Stem	Reaction
channel1	Channel	
channel2	Channel	
subscription1	Subscription	
|_ subscriber	subscriber1	
subscription2	Subscription	$\rho.stem \leftarrow message$
|_ subscriber	subscriber1	
Initialize	Cell	$\circ.channel1.subscriptions.add \leftarrow \circ.subscription1$
		$\circ.channel2.subscriptions.add \leftarrow \circ.subscription2$
Run	Cell	$\circ.channel1 \leftarrow \circ.Literal.String."HelloWorld"$
		$\circ.channel2 \leftarrow \circ.Literal.String."HelloWorld"$

3.3.1 Object Model

The class diagram in Figure 3.11 shows the classes and their associations used to implement the object model described in Section 3.1. Each class is described roughly in the following list and their functionality in the following sections.

Cell Implements compositional cell hierarchies with each instance referencing its parent and children objects. The tree may be incomplete since children are loaded on-demand but an upwards branch is always completely loaded.

Deliverer Interface for classes that are able to deliver a message such as `Cell` and `Peer`. The latter is described in Section 4.2.

Reaction Implements the reaction of the programming model which executes a list of mailings.

NativeReaction An abstract type for reactions that may execute any kind of code. Being able to perform computations without sending messages, subclasses of this class break the otherwise endless recursion of message passing.

Mailing Stores paths of receiver and message cells.

Path Represents a cell path as a list of strings, each string being a cell name.

Messenger Every mailing is delivered by its own messenger instance which spawns a new thread an re-tries sending its message until the delivery returns a positive result.

Delivery Holds parameters which are stacked for nesting deliveries (see Section 3.3.6).

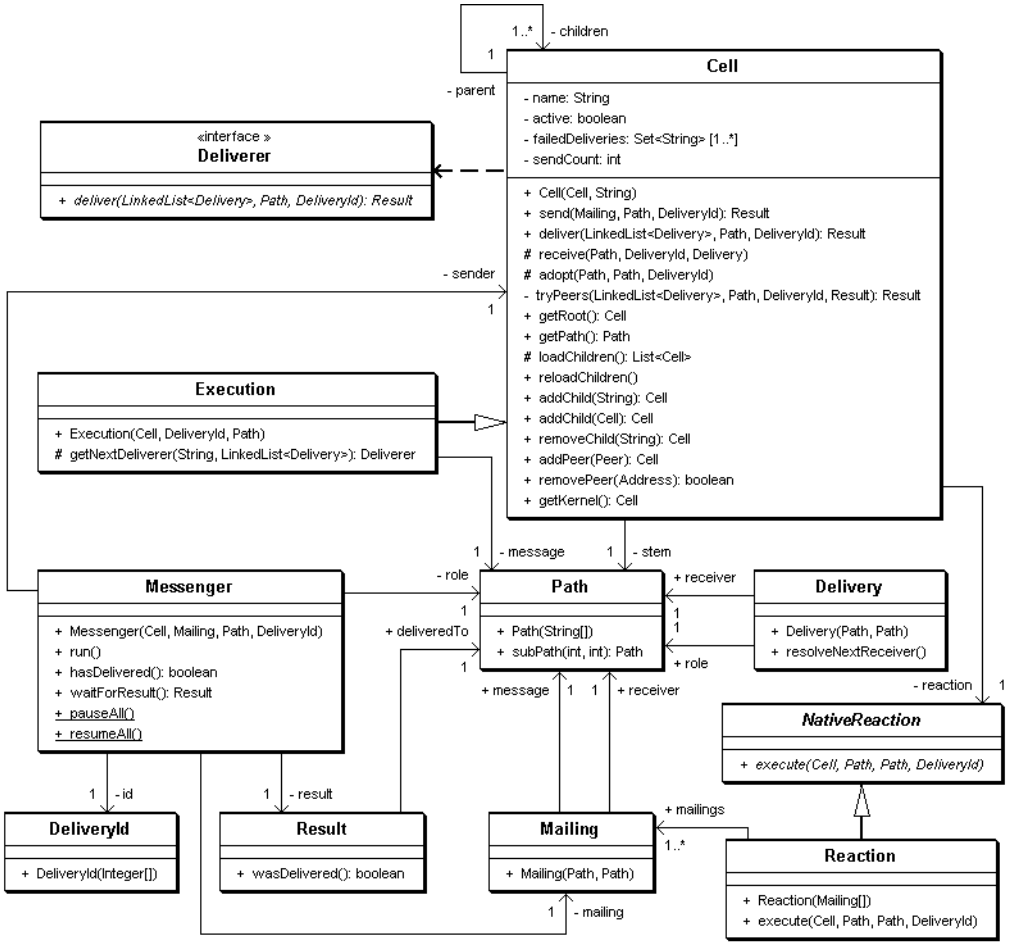


Figure 3.11: Class diagram of object model.

DeliveryId Unique identifier to avoid circular deliveries.

Result Return value of deliveries to determine success and return logged information.

Execution Class of execution cell objects which is created for every received message. Contains local cells and resolves references to its message alias.

3.3.2 Message Passing

The reaction of a cell is executed each time the cell receives a message. This is done by invoking the method `execute()` which is declared by `NativeReaction` and implemented by `Reaction`. Listing 3.1 shows the pseudo-code description of the implementation.

The method processes a list of mailings. First, the *role* of the execution is inserted at the beginning of the receiver and message paths. The role is the path of a cell as resolved during binding. It may differ from a cell's actual path due to inheritance as illustrated in Section 3.3.6. Thus by inserting the role at the beginning of the relative path, it is made absolute. A message is sent using a new instance of `Messenger`.

Listing 3.1: Execute method of Reaction

```
1 void execute (receiver, role, message, id) { {
2   foreach (mailing in mailings) {
3     mailing.receiver.insert(0, role);
4     mailing.message.insert(0, role);
5
6     new Messenger(receiver, mailing, role, id).start();
7   }
8 }
```

The class `Messenger` spawns a new thread when instantiated in which the message will be re-sent until it is delivered successfully. This is done by the method `run()` of `Messenger` which is described in Listing 3.2. In this method, a new delivery stack is created with a single delivery containing the role of the execution and the receiver path. A unique delivery identifier is created which is used to detect and avoid endless delivery loops. The resolution of the receiver cell is started by invoking the `deliver()` method of the sender cell.

Listing 3.2: Run method of Messenger

```
1 void run() {
2   do {
3     if (!pausedAll) {
4       var deliveryStack := new DeliveryStack();
5       deliveryStack.add(new Delivery(role, mailing.receiver));
6
7       var uid := new ExecutionId(eid, sender.count);
8       sender.count := sender.count + 1;
9
10      result := sender.deliver(deliveryStack, mailing.message, uid);
11    }
12  } while (!result.wasDelivered());
13 }
```

3.3.3 Binding

All references, even those to stem cells and enclosing cells, are late bound. The object model provides the structure for binding the path of a receiver cell to a reaction. This is done recursively by the method `deliver()`, defined in the `Cell` class. Each invocation

of the method resolves a child of the current cell which may be inherited or located on remote sites. Listing 3.3 contains a pseudo-code description of the `deliver()` method.

The binding of a receiver path to a reaction begins with the sender cell which is the original receiver. The instance of `Messenger` processing the mailing passes the created `deliveryStack`, the path of the `message` cell and a unique execution identifier (`id`) to the `delivery()` method of the sender cell object.

The method starts with popping deliveries that have reached their receiver off the stack as explained in more detail in Section 3.3.6. The execution identifier is used in line 7 to avoid endless delivery loops caused by circular structures of stem cells or distributed cells. If a certain receiver was resolved by the same cell in the same delivery before, the resolution step fails by returning an empty `Result`.

Lines 10 and 11 check whether the current cell is the receiver and also contains a reaction. If so, the cell's reaction is executed by the auxiliary method `executeReaction()` which is described in Section 3.3.4.

If the current cell is not the receiver, the next cell in the receiver path has to be found. This is done locally in line 16 using `getNextDeliverer()`. The method first checks if the next reference is an alias and resolves it as described in Section 3.3.5. If it is not an alias, the child is searched for in the list of the cell's own children and returned.

If no next deliverer was found this way, the cell cannot resolve it locally and the delivery is forwarded to distributed parts of the cell by the method `tryPeers()`. The method and other details of distribution are described in Section 4.2. If the receiver was found on a remote system, the delivery returns the positive result in line 23.

Otherwise, line 27 checks whether a next deliverer was found previously. If not, and a stem cell path is defined, the child or reaction is inherited by forwarding the delivery to the stem cell. This is done by pushing a new delivery to the stack as described in Section 3.3.6.

In line 33 the delivery is continued with the next deliverer that was found suitable during the algorithm and a possibly modified delivery stack. If the receiver was successfully inherited and is a child of the current cell, it is adopted in line 38 which is described in Section 3.3.7. If no next deliverer was found, nor can the receiver be inherited because no stem cell path is defined, the delivery fails in line 44.

3.3.4 Execution

As described in Section 3.1.6, the receiver cell does not execute the reaction itself but creates a child cell to do so. This is done by the method `executeReaction()` which is

Listing 3.3: Binding algorithm

```
1  Result deliver (deliveryStack, message, id) {
2    deliveryStack.popCompletedDeliveries();
3
4    var delivery := deliveryStack.first;
5    var nextCell := nil;
6
7    if (searchedBefore(id, delivery.receiver))
8      return new Result();
9
10   if (delivery.receiver.isEmpty()) {
11     if (reaction ≠ nil) {
12       executeReacion(message, id, delivery);
13       return new Result().deliveredTo(delivery.role);
14     }
15   } else {
16     nextCell := getNextDeliverer(deliveryStack);
17   }
18
19   if (nextCell = nil) {
20     addToSearchedBefore(id, delivery.receiver);
21     var peerResult := tryPeers(deliveryStack, message, id);
22     if (peerResult.wasDelivered())
23       return peerResult;
24   }
25
26   var inherited := false;
27   if (nextCell = nil and stem ≠ nil) {
28     inherited := true;
29     nextCell := this;
30     deliveryStack.push(new Delivery(delivery.role, stem));
31   }
32
33   if (nextCell ≠ nil) {
34     var nextResult := nextCell.deliver(deliveryStack, message, id);
35     if (nextResult.wasDelivered()) {
36       if (inherited
37           and nextResult.deliveredTo.contains(getPath())) {
38         adopt(inheritedPath(nextResult), message, id);
39       }
40       return nextResult;
41     }
42   }
43
44   return new Result();
45 }
```

Listing 3.4: Execution of reaction

```
1 void executeReaction (message, id, delivery) (
2   var executionName := "#" + id;
3   if (delivery.role = getPath()) {
4     addChild(new Execution(this, executionName, message));
5   }
6   delivery.role.add(executionName);
7
8   reaction.execute(this, delivery.role, message, id);
9 }
```

invoked in line 12 of Listing 3.3. The pseudo-code description of the method is shown in Listing 3.4 .

The execution cell is created as a child of the receiver with a unique name composed of the delivery identifier. If the current cell was inherited (role does not equal path), the execution is not added and has to be adopted by the inheriting cell which is described in Section 3.3.7. The stem cell path of the execution cell is set to parent by its constructor. In the last line, the cell's reaction is executed under the context of the execution cell.

The message alias of the execution cell is resolved by `Execution` by overriding the `getNextDeliverer()` method as shown in Listing 3.5. The method checks if the next cell to be resolved has the name "message" and if so, removes it from the receiver path and inserts the message cell path instead.

Listing 3.5: Resolution of message alias

```
1 Cell getNextDeliverer(deliveryStack) {
2   var delivery = deliveryStack.first;
3   var name := delivery.receiver.first;
4
5   if (name = "message") {
6     delivery.receiver.removeFirst();
7     delivery.receiver.insert(0, message);
8     return this;
9   } else {
10    return super.getNextDeliverer(deliveryStack);
11  }
12 }
```

3.3.5 Aliases

In line 16 of the `deliver()` method in Listing 3.3, the next deliverer is searched for within the current cell by invoking `getNextDeliverer()`. The receiver is found if

it is either an alias or a child of the cell. Aliases differ from children in their effect on the role of the next deliverer as shown by the pseudo-code description of the `getNextDeliverer()` method in Listing 3.6.

First, the method checks the name of the next cell in the receiver path against the three aliases (∘ (root), parent and self) and modifies receiver and role paths if a match is found. If the name equals stem, the delivery is forwarded to the stem cell but the role is not changed which leads to a behaviour similar to *super* in conventional object-oriented languages. As a last step, the next deliverer is searched within the children of the cell.

Listing 3.6: Alias and child resolution

```
1  Cell getNextDeliverer(deliveryStack) {
2    var delivery := deliveryStack.first;
3    var name := delivery.receiver.first;
4
5    if (name = "∘") {
6      delivery.receiver.removeFirst();
7      delivery.role := createPath("∘");
8      return getRoot();
9
10   } else if (name = "parent") {
11     delivery.receiver.removeFirst();
12     delivery.receiver.insert(delivery.role.subPath(-1));
13     return this;
14
15   } else if (name = "self") {
16     delivery.receiver.removeFirst();
17     return this;
18
19   } else if (name = "stem") {
20     delivery.receiver.removeFirst();
21     deliveryStack.push(createDelivery(delivery.role, stem));
22     return this;
23
24   } else {
25     var child := getChild(name);
26
27     if (child ≠ nil) {
28       delivery.role.add(delivery.receiver.removeFirst());
29       return child;
30     }
31   }
32
33   return nil;
34 }
```

3.3.6 Inheritance

Cells are inherited by redirecting the delivery to the cell's stem cell, if defined. Since the stem cell path has to be resolved using a delivery as well, deliveries are stacked to store the role and receiver paths of previous deliveries. When the new delivery reached its receiver (the stem cell), the previous delivery is restored by popping the arrived delivery off the stack (see line 2 in Listing 3.3) which results in the stem cell using the role of the inheriting cell.

An example of stacked deliveries is shown in Figure 3.12 using two levels of inheritance. In the example, a message is sent by the root to the inherited cell A.X. The delivery reaches A which does not contain X but defines ∘.B.D as its stem cell. To resolve the stem cell, a new delivery is created in Step 3 and pushed on the stack. The stem cell D is itself inherited from C and therefore requires a third delivery in Step 6. C is reached in Step 8 and the nested delivery popped off in Step 9, which restores the role ∘.B of the inheriting cell. This is repeated for the initial delivery in Step 11, which restores the role of the original receiver so the final role resolves to ∘.A.X.

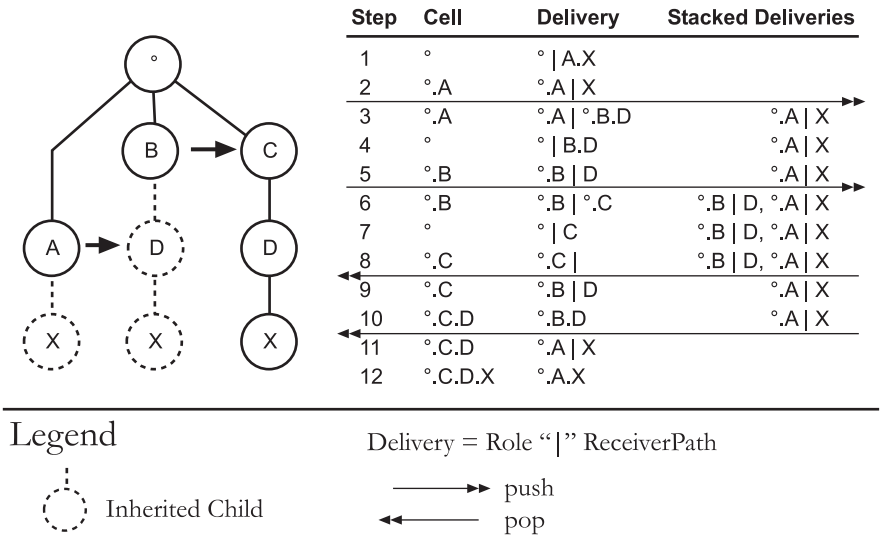


Figure 3.12: Example of nested deliveries during resolution of an inherited cell.

3.3.7 Adoption

The binding algorithm implements a copy-on-write strategy and creates new children when a message is sent to an inherited cell in order to store the execution cell. This procedure is called *adoption*. Line 3 of Listing 3.4 makes sure that the execution cell is not added to the children of an inherited cell and line 38 of Listing 3.3 invokes the method `adopt()` if a receiver was inherited and is offspring of the current cell.

Listing 3.7 describes the `adopt()` method in pseudo-code. Its argument `inherited-Path` contains the part of the receiver path that was inherited by the current cell. The method iterates trough the path and creates all cells in it as a child of the preceding cell. The stem of each cell is the formerly inherited cell, thus the child with the same name of the parent's stem cell.

Listing 3.7: Method to adopt executions of inherited cells

```
1  void adopt(inheritedPath, message, id) {
2    var cell := this;
3
4    foreach (name in inheritedPath) {
5      var child := new Cell(cell, name);
6
7      var stem := cell.getPath();
8      stem.add("stem", name);
9      child.setStem(stem);
10
11     cell = cell.addChild(child);
12   }
13
14   cell.addChild(new Execution(cell, eid, message));
15 }
```

Figure 3.13 illustrates the adoption of a cell. Because the adopted cell specializes the formerly inherited cell, the local copy only needs to contain the modified information. If for example the reaction of cell B.E.F was changed, it still inherits the children from C.E.F.

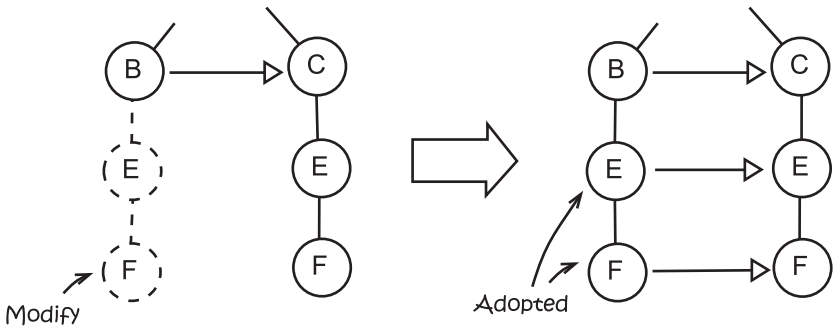


Figure 3.13: Recursive adoption of a second degree inherited cell.

Chapter 4

Environment

The previous chapter described the programming model which forms the kernel of the presented software platform. In order to develop applications, further components are needed such as a parser for descriptions of behaviour, connection with local and remote systems and library support for modification of cells and common programming tasks. Unlike the programming model, the components described in this chapter are considered interchangeable.

4.1 Storage

Cells are stored persistently using the file system by mapping a folder tree directly onto the cell hierarchy. A new cell tree is created by creating a new folder which contains the definition of the root cell. Cell definitions consist of a file "<cellName>.cell" which specifies the properties of a cell, and a folder "<cellName>" which contains the definitions of its child cells. Figure 4.1 shows an exemplary folder tree and its corresponding cell tree.

4.1.1 Syntax

The syntax presented in this section can be used to describe reactions of cells. The production rules of the syntax are given in Extended Backus-Naur Form where brackets represent options, bracelets repetition and parentheses groups.

As described in Section 3.1.4, a reaction consists of a list of mailings, each containing the paths of the receiver and message cells. In their textual description, mailings are separated by new line characters (LF and/or CR) and the paths by white-space

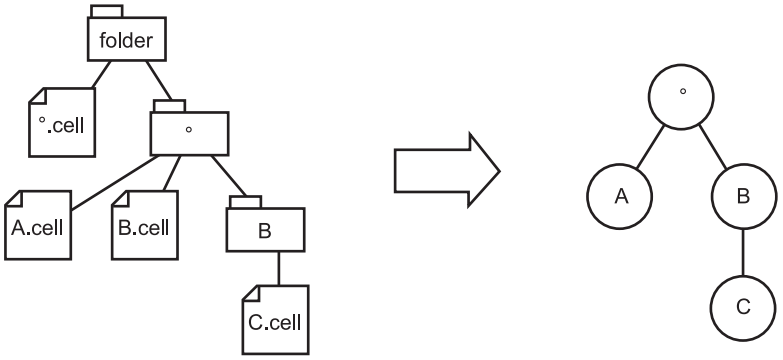


Figure 4.1: Mapping files and folders onto a cell tree.

$$
\begin{gathered}
Reaction = [Mailing\ \{NL\ Mailing\}] \\
Mailing = Receiver\ \_\ Message \\
Receiver = Path \\
Message = Path \\
Path = Name\ \{\text{'.'}\ Name\}
\end{gathered}
$$

Figure 4.2: Production rules for syntax of reaction and paths.

characters. Each path is a list of cell names (including aliases) separated by dots. This leads to the production rules of Figure 4.2.

Cell names can consist of any combination of printable and non-printable characters. Names containing a reserved character such as dots, spaces and new-line have to be quoted. Inside quoted names, backslashes must be used to escape quotation marks and backslashes. Thus, names containing backslashes or quotation marks also have to be quoted. The production rules for cell names are given in Figure 4.3.

$$
\begin{gathered}
Name = Unquoted \mid Quoted \\
Unquoted = \{\text{ANY} - Reserved\} + \\
Quoted = [Unquoted]\ QuotedPart\ [Unquoted] \\
QuotedPart = \text{''}\ \{(\text{ANY} - Escaped) \mid (\text{'\textbackslash'}\ Escaped)\} +\ \text{''} \\
Reserved = \_ \mid \text{'.'} \mid Escaped \\
Escaped = \text{''} \mid \text{'\textbackslash'} \\
\_ = \{\text{' '} \mid \text{TAB}\} + \\
NL = \{\text{LF} \mid \text{CR}\}+
\end{gathered}
$$

Figure 4.3: Production rules for syntax of cell names.

4.1.2 File Format

A cell definition file has the same name as the cell plus the extension ".cell". It uses the Extensible Mark-up Language (XML) to define the cell's stem path and its reaction. The involved XML elements are the root element `cell` with its optional sub-elements `stem` and `reaction` which contain the path of the stem cell and the reaction, respectively. The stem path and the reaction are described using the syntax of the previous section.

The `cell` and `reaction` elements have an optional boolean attribute `native` (which defaults to `false`) which indicates, if set to `true`, that the cell or its reaction is defined on kernel level and not in the definition file. Listing 4.1 shows an exemplary complete cell definition file.

Listing 4.1: Cell definition file

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <cell native="false">
3   <stem>°.Path.Of.StemCell</stem>
4   <reaction native="false">
5     First.Receiver  First.Message
6     °.Another.One   And.Its.MessageCell
7   </reaction>
8 </cell>
```

4.1.3 Implementation

Classes involved in loading, parsing and storing of cell definitions in files are shown in the class diagram of Figure 4.4. A cell hierarchy uses a single instance of `CellLoader` which is connected with the folder that contains the hierarchy, passed as its constructor's argument. The `getChild()` method is used to load the root cell.

All further cells are loaded on demand during path resolution. The first time a cell has to resolve a child, it loads its children using the `getChildren()` method. An XML parser not contained in the diagram parses and assembles cell definition files. Paths and reactions are parsed by `PathFormat` and `ReactionFormat` respectively.

If a cell or its reaction is marked *native* in the cell definition file, the `CellLoader` searches in loaded libraries for a class whose name matches the cell path. The mapping of cell paths to class names has therefore to be unambiguous.

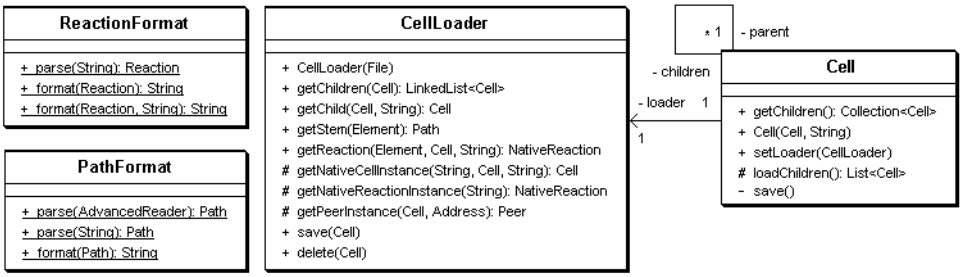


Figure 4.4: Class diagram of classes involved in cell loading and parsing.

4.2 Distribution

A requirement of the software platform is completely transparent distribution. Cells exist in a global virtual space where each cell can be reached using its path regardless on which host of a network it actually exists. Thus cells can also migrate arbitrarily within a network.

This chapter describes the architecture of distributed cell systems, how these are configured, stored and the implementation of the distributed binding algorithm.

4.2.1 Architecture

Cells of different hosts are connected using an unidirectional peer-to-peer architecture. Each cell can be connected to a number of peer cells, which are cells in the same position of a cell hierarchy on different hosts. Figure 4.5 shows an exemplary connection between cells on three different hosts.

A cell consists therefore of the union of itself and all its peers, directly or indirectly connected. Cell properties such as stem cell paths, reactions, children and peers may be distributed, moved and replicated arbitrarily throughout the distributed system. In the example in Figure 4.5, the complete list of children of A is B, C, E and F which are all reachable from all hosts. Due to redundancy of peer connections in the example, cell A.C.D of host 2 is reachable in four ways from host 1.

4.2.2 File Format

Cell definition files can be extended by `peer` elements to configure connections between cells. Each peer connection is defined by a `network` and a `host` element which contain a network identifier and the address of the host within this network. Listing 4.2 shows the definition file of a cell with one peer connected over a TCP/IP socket.

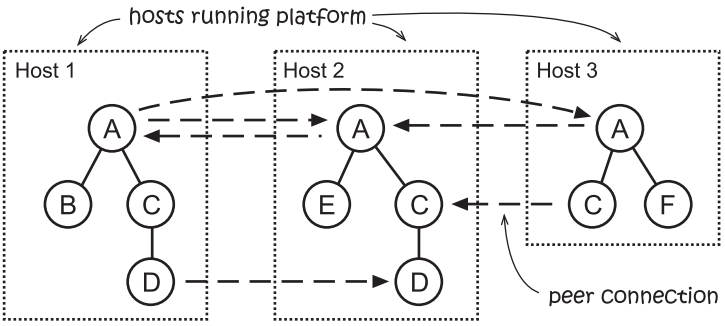


Figure 4.5: Architecture of distributed cells using unidirectional connections between peers.

Listing 4.2: Cell definition file with peer definition

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <cell>
3   <peer>
4     <network>Socket</network>
5     <host>localhost:42</host>
6   </peer>
7 </cell>
```

4.2.3 Implementation

Peer connections are implemented using a client-server architecture. Figure 4.6 shows a class diagram of the classes used in the current version. The root of a cell hierarchy is associated with one instance the abstract class `Server` for each network it is connected with. Clients are implemented by subclasses of the abstract class `Peer`. Subclasses of `Server` and `Peer` correspond to different networks and are therefore parallel as demonstrated by the `SocketServer` and `SocketPeer` classes which connect over a TCP/IP socket.

The factory method `create()` of `Peer` creates new peers based on the network identifier contained in the `Address` argument. The host address is parsed by the subclass, for example "localhost:42" is parsed by the constructor of `SocketPeer` into the host name "localhost" and port 42.

If a child cell was not found locally within its parent, the method `tryPeers()` is invoked in line 21 in Listing 3.3 which forwards the delivery to the peers of the current cell and all parents.

The `deliver()` method of `SocketPeer` transmits the path of the current cell, the delivery stack, the message path and the delivery identifier to the host specified by the

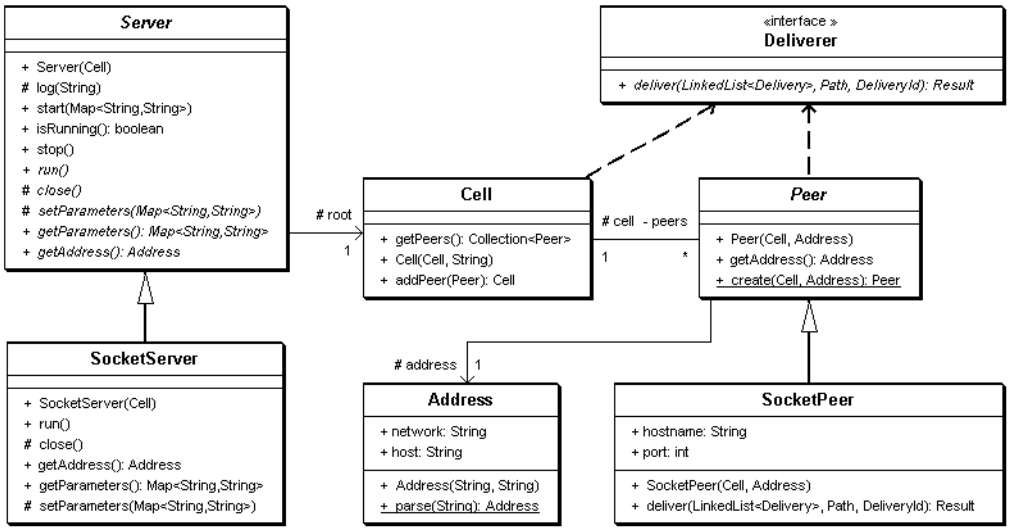


Figure 4.6: Class diagram of classes involved in distribution.

peer instance. The server spawns a new thread, resolves the path of the current cell and invokes it `deliver()` method with the transmitted arguments. If the connection times out or the peer cell on can not be resolved by the server, the cell is regarded as not existing.

4.3 Kernel Cells

Every cell has a child named cell which contains a set of cells called *kernel cells* that reflect all of the cell's properties. These cells can be used to dynamically change the cell's name, its reaction, add or remove children or peers and so on. Kernel cells cannot be deactivated thus the properties of a cell can always be read and modified, even if it is deactivated. Figure 4.7 shows composition and specialization of all implemented kernel cells in an entity-relationship diagram.

Note that the child cell is not inherited but implemented as an implicit child. Therefore it has to be handled as a special case by the binding algorithm regarding resolution and adoption which is not included in the listings of Section 3.3. It's resolution is recognized by `getNextDeliverer()` which loads the corresponding `Cell` object using the `getKernel()` method of `Cell`. This is necessary since kernel cells need local access to their target cell and reactions of inherited cells may execute on a different host than the inheriting cell.

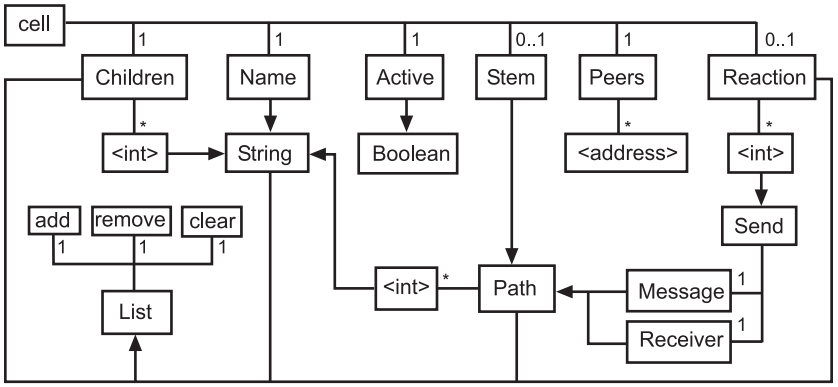


Figure 4.7: Entity-relationship diagram of cells that can be used for dynamic modification of cell properties. Arrows indicate "is-a" relationships; lines with multiplicities indicate "has-a" relationships.

4.4 Library

The kernel of the software platform only provides mechanisms for distributed and concurrent message passing. All further components such as data types and control structures are part of a cell library. The following sections contain descriptions of all library cells implemented in the current version of the software platform. It is a minimal implementation consisting only of cells necessary for the applications described in Chapter 5 and and Section 6.3. The path of the stem cell is given after a colon unless it is ∘.Cell. All cell paths are relative to the root cell.

4.4.1 General

Cell Default stem and and root of specialization hierarchies.

Cell.respond Creates a new cell response as child of its parent with the receiver cell as its stem.

Cell.equals Considers the message cell equal to its parent cell if it contains the same child cells and all of the children consider themselves equal as well. The cell can be overridden to implement a domain-specific definition of equality.

Zells Container of the generic cell library.

4.4.2 Data Types

Zells.Boolean Abstract basic boolean value.

Zells.Boolean.or Abstract cell for the boolean OR operation.

Zells.Boolean.and Abstract cell for the boolean AND operation.

Zells.True : Zells.Boolean Represents the boolean true value.

Zells.True.or Responds True.

Zells.True.and Responds with the received message.

Zells.False : Zells.Boolean Represents the boolean false value.

Zells.False.or Responds with the received message.

Zells.False.and Responds False.

Zells.Number Abstract type for numbers.

Zells.Number.equals Considers the message cell equal to its parent if it specializes a number with the same value.

Zells.Number.add Replies the sum of its parent and the message cell (must specialize a literal number).

Zells.Number.subtract Subtracts the message cell (must specialize a literal number) from its parent.

Zells.String : Zells.List Basic abstract stem cell for strings. A string is a list of characters.

Zells.Character Basic abstract stem cell for characters.

Zells.Literal.Number Parent of all literal numbers, negative and positive. Floating point numbers are children of literal numbers. Examples are Zells.Literal.Number.42, Zells.Literal.Number.-5 and Zells.Literal.Number.5.23.

Zells.Literal.String Parent of all literal strings which are lists of literal characters. Examples are Zells.Literal.String."Hello World" and Zells.Literal.String.Test. Note that alias cells result in literal strings as well, e.g. Zells.Literal.String.parent corresponds to the string "parent" rather than the cell Zells.Literal.String.

Zells.Literal.Character Parent of literal characters such as Zells.Literal.Character.A and Zells.Literal.Character.?.

Zells.List Basic abstract type for all numbered lists. Elements are children of the list with the index number as their names.

Zells.List.add Adds and element to the end of the list which specializes the message cell and replies the new element.

Zells.List.remove Removes element at position of message cell which specializes a literal number and replies the removed element. The indices of all following elements are decremented.

Zells.List.clear Removes all elements of the list.

4.4.3 Control Structures

Zells.Boolean.ifTrue Has an empty reaction.

Zells.Boolean.ifFalse Has an empty reaction

Zells.True.ifTrue Executes the reaction of its message by sending it a message.

Zells.False.ifFalse Executes the reaction of its message by sending it a message.

Zells.List.each Sends all of its parent's children to the message cell.

4.4.4 Reflection

Cell.cell Container of all reflection cells.

Cell.cell.create Provides a short-cut to for creating new cells. By sending a message to one of its children, it creates a new cell with the name of the child and the message as stem cell.

Cell.cell.Name : Zells.String Specializes literal string that corresponds to name of cell. Only if the its stem is changed directly, the cell's name is changed.

Cell.cell.Stem : Zells.Reflection.Path Specializes a Path that reflects the stem cell path. Its stem cell can not be changed, thus it has to be modified directly to change stem cell.

Cell.cell.Reaction : Zells.List Reflects reaction of cell as list of Sends. Has to be directly modified as well.

Cell.cell.Children : Zells.List Reflects the cell's own children (no inherited nor distributed children) as a list of their names.

Cell.cell.Active : Zells.Boolean Reflects the activation status of the cell. Stem has to be modified directly to change state.

Cell.cell.Peers Contains list of connected peers, each with its address as name. Elements of this list are cells which only access their local parts and do not forward deliveries to other peers.

Zells.Reflection.Path : Zells.List Represents a cell path as list of strings.

Zells.Reflection.Mailing Abstract cell to represent mailings which structure receiver and message paths.

Zells.Reflection.Send.Message : Zells.Reflection.Path Specializes the path of the message cell.

Zells.Reflection.Send.Receiver : Zells.Reflection.Path Specializes receiver cell path.

Chapter 5

Development Tools

The prototype implementation of the described software platform is available at http://zells.org. It is built with Java SE 6 and includes graphical development tools for editing, monitoring and analysing distributed cell systems. The following sections describe the usage of the provided tools and their implementation.

5.1 Description

The development tools consist of several components to analyse cell systems on different levels. The *message sender* enables the user to send messages to cells transparent to inheritance and location, the *cell browser* edits and shows only actually existing cells and their distribution, the *message inspector* visualizes messages and consecutive messages, and the *delivery analyser* traces the path of individual deliveries. The following subsections describe these components individually.

5.1.1 Message Sender

The main window as shown in Figure 5.1 consists of a menu and a form to send messages. The menu lets the user load cell systems from folders containing definition files (see Section 4.1, start and stop servers needed for distribution and show the cell browser.

The form can be used to send messages with the root cell as sender. To send a message, the paths of the receiver and message are entered in the corresponding text fields and the button "send" is clicked. All paths are resolved relative to the root.

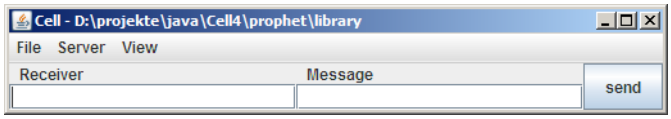


Figure 5.1: Main window and form to send initial messages.

5.1.2 Cell Browser

The interface also provides a graphical representation of a distributed cell system, a cell browser, which enables the user to browse and modify existing cells, create new cells and select cells for message sending.

Cells can be edited using a context-menu or by dragging it inside another host to copy it, move it or create new peer connections. Figure 5.2 shows an exemplary browser with two hosts. Cells can be extended and collapse to show and hide their children, indicated by plus and minus signs on the bottom left corner of each cell. Only actually existing and no inherited cell are shown. Little triangles and circles on the right side indicate defined stem cell paths and reactions respectively.

The darker filling of cell ∘.B.B in the figure indicates that it is an inactive cell. Inactive cells behave like non-existent but allow to set and change their properties such as name, reaction, stem and children. This way, a cell can be completely defined before it is visible to other cells.

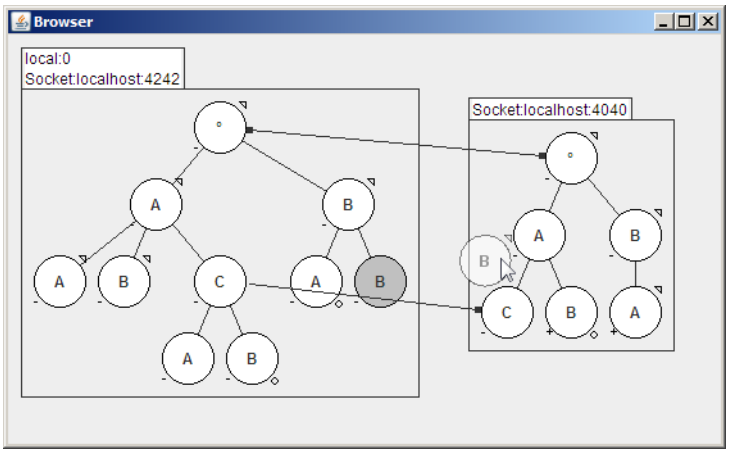


Figure 5.2: Browser for distributed cell systems.

Cells are connected directly to corresponding peers on other hosts. These connections are unidirectional, possibly redundant and make all children of the peer and its peers addressable for the cell and all of its children. Peer connections are represented by lines between cells on different hosts with a black circle indicating the direction of

the connection. Hosts are identified using the network identifier and the address of the host within this network.

5.1.3 Message Inspector

Every time the message sender (Figure 5.1) is used to send a new message, a message inspector opens to show all resulting sends as a tree of mailings. Along with the sender, the receiver and the message, the inspector indicates the current status of each mailing which can be "Sending" if the receiver cell is being resolved, "Waiting" if the receiver cell was not found and the message is re-sent, "Paused", "Cancelled" or "Delivered to ..." as shown in the exemplary message inspector in Figure 5.3.

Sends can be paused, resumed and cancelled individually or paused and resumed globally. This allows the user to freeze the system in order to analyse deliveries as described in the following section.

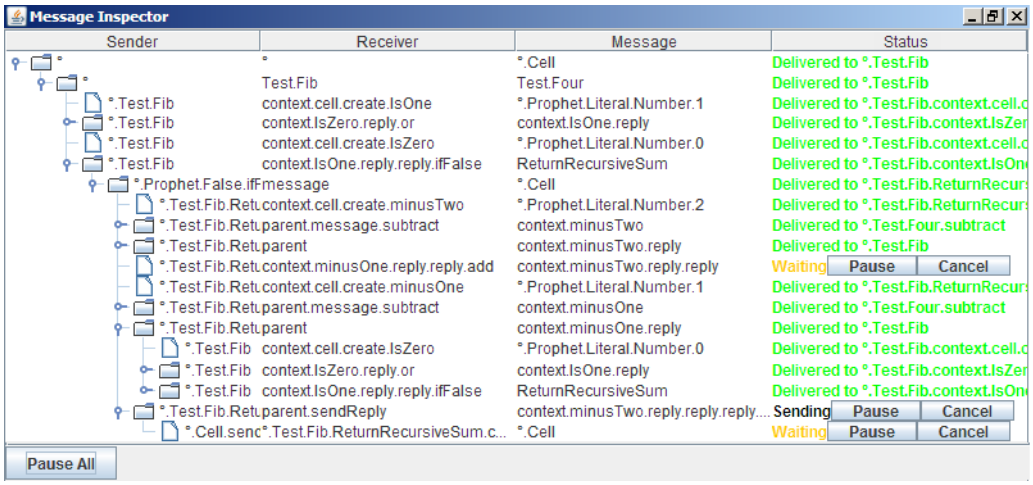
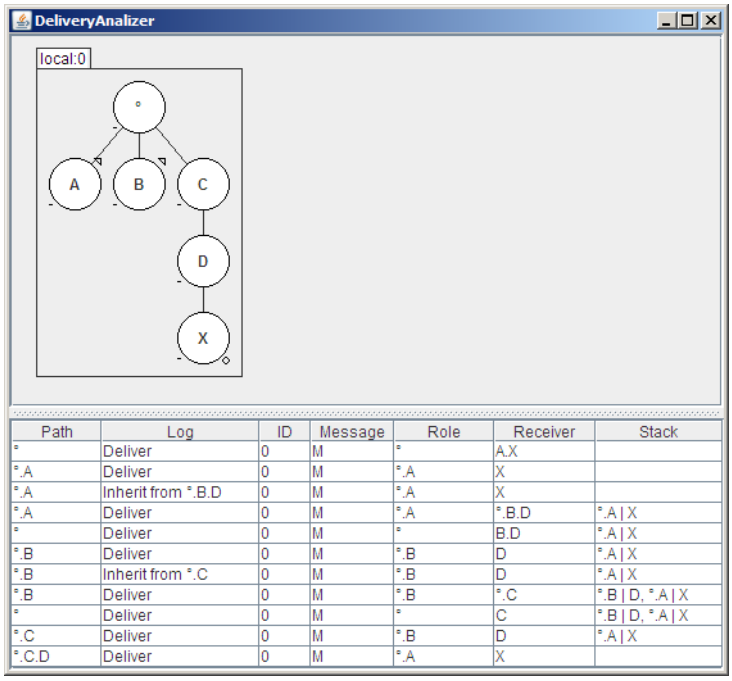


Figure 5.3: Message inspector showing a tree of mailings.

5.1.4 Delivery Analyser

By double-clicking on a mailing in the inspector, a delivery analyser opens and visualizes the delivery's log. The log contains entries for the steps of the name resolution with information about the path of the delivering cell and the delivery parameters for each step as illustrated in Figure 5.4. By clicking on a log entry, the current and all previous delivering cells are highlighted in the cell browser above the log. This way it is easy to follow the path of the name resolution and identify reasons for unsuccessful deliveries.



Path	Log	ID	Message	Role	Receiver	Stack
°	Deliver	0	M	°	A.X	
°.A	Deliver	0	M	°.A	X	
°.A	Inherit from °.B.D	0	M	°.A	X	
°.A	Deliver	0	M	°.A	°.B.D	°.A | X
°	Deliver	0	M	°	B.D	°.A | X
°.B	Deliver	0	M	°.B	D	°.A | X
°.B	Inherit from °.C	0	M	°.B	D	°.A | X
°.B	Deliver	0	M	°.B	°.C	°.B | D, °.A | X
°	Deliver	0	M	°	C	°.B | D, °.A | X
°.C	Deliver	0	M	°.B	D	°.A | X
°.C.D	Deliver	0	M	°.A	X	

Figure 5.4: Delivery analyser visualizing a delivery log.

5.2 Implementation

As the kernel, the graphical development tools are implemented using Java SE 6. But since only message passing is used to access the cells, the tools could be implemented using any technology including the presented software platform itself. This section describes how the cell browser uses sending messages to kernel cells (see Section 4.3) to read and modify properties of distributed cells.

The cell browser needs to be able to access and modify remote cells. This could be done (and was done initially) using remote method calls on the kernel level to directly access `Cell` objects. Besides the cost of making all property setters and getters available to remote invocation, this approach also has the disadvantage that it only works with directly reachable cells. But as described in Section 4.2, cells can be connected indirectly through different networks. In such a case, the connecting cells play the role of inter-network relays and no direct access to the remote methods is possible.

For these two reasons and also because of the increased portability, the browser uses only the message passing mechanism to access all cells, local and remote. Note that the kernel cell cell.Children only contains the list of the cell's own, i.e. neither inherited nor distributed children, so the browser only shows actually existing cell.

Another kernel cell cell.Peers provides the possibility to disable network transparency. If a message is sent to a cell A on host "n:1" is accessed using A.cell.Peers.n:1,

it is guaranteed to be delivered to the peer on host "n:1" and to no other cell. Also, messages sent to children of A using this path are only delivered to children on the given host.

The cell browser loads and accesses child cells incrementally during browsing. The properties of each cell are read when the cell is loaded including peer connections which leads to an incremental discovery of new hosts. Figure 5.5 illustrates this process in three exemplary steps.

Step (a) shows a new browser instance with the local host containing the cell A.

In step (b) A is expanded and its child B loaded which has one peer connection with host "1" on network "n". The path to access the peer is therefore A.B.cell.Peers.n:1. Its parent cell A is assumed but cannot be accessed since no connection exists.

In step (c) Cell A.B on host "n:1" is expended, loading its child C which has a peer connection with host "A" on network "m". The new peer is accessed using the path A.B.cell.Peers.n:1.C.m:A. As before, its parent cells can be assumed but not accessed. Note that the local host has no access to network "m" so the only way to reach cell C on host "m:A" is over cell B on host "n:1".

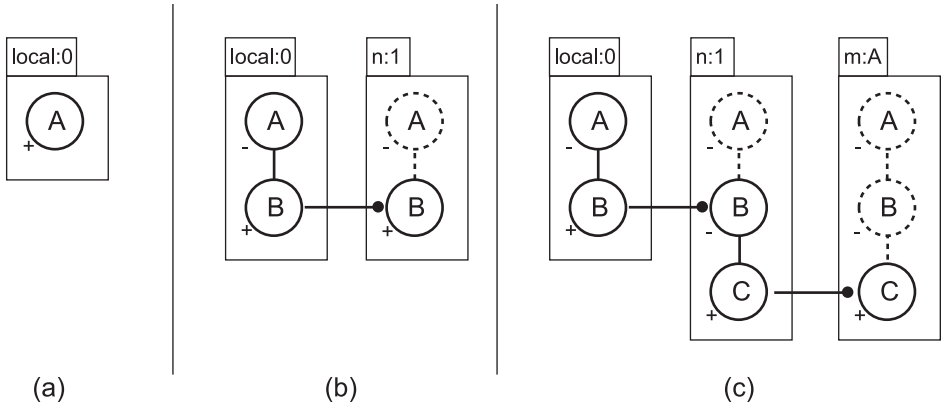


Figure 5.5: Discovery of peers on new hosts and their implicit parents in cell browser. **(a)** A new cell browser showing one cell in local cell hierarchy. **(b)** Cell A is expanded, its child B has a connection with its peer on host "n:1". **(c)** A new peer is discovered on host "m:A".

Chapter 6

Discussion

The following sections discuss the results of this project by comparing it with related works and describing reasons of important design decisions. The last section describes experiences with tests and experiments during the development.

6.1 Related Work

The presented programming model combines many features of related models and languages. Being object-oriented, it is related to Smalltalk [1, 23], C++ [24] and Java [25]. Compared to the hybrid models of C++ and Java, the presented model follows the object-oriented paradigm more consistently. None of these platform are inherently distributed and concurrent although they include frameworks that provide these features.

Newspeak [26, 27] extends the semantics of Smalltalk with the ability to nest classes in order to build modules. Its semantics are therefore similar to the presented programming model with the difference that there is no global address space in Newspeak and like Smalltalk it is not distributed.

A prominent programming model which is closely related to the presented one is the actor model [28, 29]. In fact, the presented software platform can be seen as an implementation of the actor model with cells playing the role of actors. Like actors, cells are distributed, thus inherently concurrent and use asynchronous messages for communication. Unlike actors, cells do not use a *become* primitive to change their behaviour since it conflicts with the use of inheritance [30].

The currently most successful implementation of the actor model is the Erlang programming language [31, 32]. The main differences with Erlang is that the proposed

model does not buffer messages in mailboxes and uses a higher level of abstraction with a single kind of entity compared to Erlang's variety of control and data structures. Since the proposed model does not specify a high level programming language, a functional language similar to Erlang could be used, although an imperative language fits more naturally to an object-oriented model.

Another implementation of the actor model is the experimental programming language Act 1 [33]. It shares the radical view of "everything is an actor" with the proposed model thus data types, procedures as well as messages are actors. Besides a different approach to distribution and name resolution, the main difference is that Act 1 uses pattern matching to bind messages to message handlers.

The programming language Oz (as used in [34]) uses a distributed and concurrent model as well and also data flow variables. But unlike the proposed model, Oz is not inherently object-oriented and thus does not include the concepts of specialization and inheritance, although they can be added.

Self [35] is known for its consistent use of prototype objects instead of classes but new objects are created as copies of prototypes and not as specializations. Also, Self is not distributed. Its model of activation records is very similar since activations are modelled as specializations of the receiving entity but the fact is hidden from the user by the special receiver "self" which behaves differently depending on if a message is sent to it directly or to a contained entity.

6.2 Design Process

The rapid prototyping approach described in Section 2.2 led to different designs of components of the programming model during the evolution of the prototypes. This section describes this process in three cases that were object of the majority of revisions.

6.2.1 Genesis

One design principle of the presented programming model is maximizing extensibility by minimizing structure. Therefore, creating new cells should be possible by only relying on the message passing mechanism.

Creating cells by sending messages conflicts in a certain way with the analogy of biological cells used in Section 3.1, where a new cell would be created in an empty space by specializing an existing cell. The new cell would then be adopted by a parent cell. In practice, it has to be done the other way around since a cell without a parent can not be addressed. Thus it is the parent's responsibility to create a new child.

The initial approach was to send a cell containing the definition of the new cell to the parent cell. This way a cell could be created with a single message. The problem was that the definition cell itself had to be created as well which would lead to an infinite regression of creating definition cell. The regression was broken by *literal* definition cells which similar to literal strings (see Section 4.4) can be addressed as the child of a native cell. The name of this child is then used by the literal definition as name for the new cell.

Although the approach was working, it required a lot of library support and duplicated all cell definitions. It was therefore discarded in favour of an approach using reflection cells. The cell Children represents the children of a cell as a list of strings. Cells can be created and deleted by simply modifying this list. The advantage is that the same structure can be used for reading and modification without redundancy.

Another problem with cell genesis is timing. Since all mailings of a reaction are sent concurrently, messages sent to a new cell might be sent before the cell was created. The first approach to use continuations for synchronization was no feasible because continuations are cells which have to be created first as well. The problem was eventually solved by introducing data flow synchronization and an activation state which allows a cell to be created and defined before it is activated as described in Section 3.1.7.

6.2.2 Responses

Due to the asynchronism of message passing, messages which expect a response have to contain a resumption cell that the receiver can send the response to. As described in Section 3.2.2, the current solution is to create a specialisation of the actual message cell which contains the resumption cell as child. The disadvantage of this approach is the overhead caused by creating a new container cell.

The overhead could be avoided by integrating resumption in the meta structure of mailings. A mailing would then consist of the paths of the receiver, the message and a third path of the resumption cell.

The reasons against making resumption cells part of mailings are rather philosophical. It conflicts with the design principles of minimizing structure and separating meaning from optimization. Not only responses could be integrated into the meta structure but also exceptions and similar constructs, bloating the kernel more than absolutely necessary.

6.2.3 Binding & Context

Most exploration was done regarding the binding of the receiver cell path to a reaction and its execution. Revised designs and implementations of these central components were therefore the reasons for most new prototypes.

This was caused by the difficulties related to managing the context of each execution in a concurrent, possibly recursive tree of mailings. As Figure 6.1 demonstrates, a single mailing may lead to the reaction of a cell being executed several times simultaneously. Therefore, the binding algorithm has to keep track of the execution context under which a cell sent a message to the alias message or cells which exists only locally within a context. The resolution of these cells depends on the current execution context.

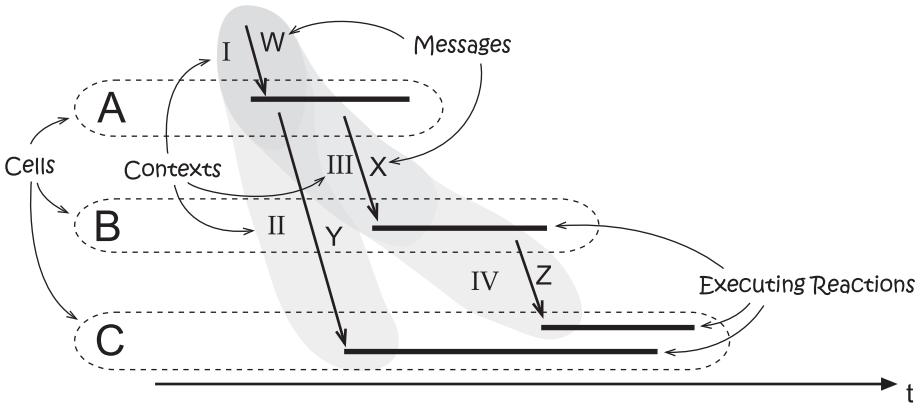


Figure 6.1: Contexts in a tree of mailings resulting in reaction of cell C being executed simultaneously in two different contexts.

In a first approach, all information of the current and all preceding mailings was passed to each resolution step which lead to a total of seven parameters, most of them stacked lists to provide nested deliveries (see Section 3.3.6). The overhead was considered necessary to be able to resolve context-dependent children on any host and also to avoid resolution loops caused by circular peer connections.

In a later version, the parameter overhead was reduced by introducing an execution identifier which was unique for every context. It consisted of the identifiers of previous executions in order to provide references to local children of these which where stored in a child of the receiver cell with the identifier as its name. This context child was not addressable directly but only by using the special cell name "context" which was resolved dynamically to the corresponding cell depending on the execution context information. Besides of the still considerable overhead, this approach was discarded since it was inconsistent with the programming model.

As described in Section 3.2, the current version reduced the overhead further by storing all execution context information in a specialisation of the receiver cell. Since the reaction is executed by this context dependent cell, no dynamic resolution of a special "context" cell is necessary and the implementation completely consistent with the programming model. Its main disadvantage is the need for the user to be aware of the execution cell in order to address children of parent cells correctly.

The introduction of this *execution* cell also caused a change of the adoption mechanism which is described in Section 3.3.7. In earlier versions, a copy-on-change strategy was implemented. Inherited cells were only adopted when explicitly modified by kernel cells. Since this process started at the bottom of the cell tree, it had to be repeated recursively until a not-inherited cell was reached. In the current version, a copy-on-receive strategy is used which leads to more adoptions but each with less overhead.

6.3 Experience

Besides the unit testing described in Section 2.3, two test applications were implemented to verify the correctness of the software platform. The two applications are the publish-subscribe system which is already presented in Section 3.2 and a recursive algorithm to calculate numbers of the Fibonacci sequence which is described in this section. Both applications were implemented as automated tests and manually using the development tools which facilitated debugging considerably.

Algorithm

The Fibonacci sequence is defined recursively as $Fib_n = Fib_{n-1} + Fib_{n-2}$ with the two initial values $Fib_0 = 0$ and $Fib_1 = 1$. The definition can be implemented as the recursive algorithm in Listing 6.1.

Listing 6.1: Recursive algorithm to calculate a number of the Fibonacci sequence

```
1 int Fibonacci(index) {
2   if (index = 0 or index = 1) {
3     return index;
4   } else {
5     return Fibonacci(index - 1) + Fibonacci(index - 2);
6   }
7 }
```

Implementation

This algorithm was chosen as test application because of its extensive use of recursion which involves local data and therefore tests the execution context described in Section 3.2 thoroughly. In practice, its implementation led to the discovery of several errors which could be identified and fixed with the help of the graphical tools. All of the errors were related to concurrency issues which increases with the index of the calculated number.

Figure 6.2 shows the cells defined for the implementation of the algorithm on the presented software platform. The reactions of the cells are presented in the following listings. See Section 4.4 for descriptions of the used library cells.

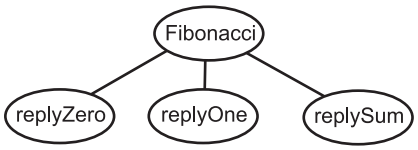


Figure 6.2: Hierarchy of cells involved in the Fibonacci example application.

To calculate the Fibonacci number with the index n, the number is sent to Fibonacci as a message. The reaction of Fibonacci shown in Listing 6.2 creates in lines 1 to 3 a container cell isOne which is tested for equality with the received index by being sent to message.equals. By sending it to ifTrue of the response, the reaction of Fibonacci.respondOne described in Listing 6.3 is executed if the message equals the number one. This is repeated in lines 5 to 7 with the number zero and the corresponding cell Fibonacci.respondZero whose reaction is shown in Listing 6.4.

Listing 6.2: Reaction of Fibonacci cell

```
1  cell.create.isOne ← ∘.Zells.Literal.Number.1
2  message.equals ← isOne
3  isOne.response.ifTrue ← respondOne
4
5  cell.create.isZero ← ∘.Zells.Literal.Number.0
6  message.equals ← isZero
7  isZero.response.ifTrue ← respondZero
8
9  isOne.response.or ← isZero.response
10 isZero.response.response.ifFalse ← respondSum
```

Listing 6.3: Reaction of Fibonacci.respondOne cell

```
1 parent.parent.message.respond ← ∘.Zells.Literal.Number.1
```

Line 9 of the reaction of Fibonacci tests if either the the responses of isOne or the response of isZero is true. If not, the reaction of Fibonacci.respondSum which is given in Listing 6.5 is executed in line 10.

The reaction of Fibonacci.respondSum is the implementation of line 5 of the algorithm in Listing 6.1. In lines 1 to 3 the index number is subtracted by one and the result sent to Fibonacci. This is repeated with the number two in lines 5 to 7. The

Listing 6.4: Reaction of Fibonacci.respondZero cell

```
1 parent.parent.message.respond ← ∘.Zells.Literal.Number.0
```

Listing 6.5: Reaction of Fibonacci.respondSum cell

```
1 cell.create.minusOne ← ∘.Zells.Literal.Number.1
2 parent.parent.message.subtract ← minusOne
3 parent.parent.parent ← minusOne.response
4
5 cell.create.minusTwo ← ∘.Zells.Literal.Number.2
6 parent.parent.message.subtract ← minusTwo
7 parent.parent.parent ← minusTwo.response
8
9 minusOne.response.response.add ← minusTwo.response.response
10
11 parent.parent.message.respond ← minusTwo.response.response.response
```

responses of lines 3 and 7 are added in line 9 whose result is sent to the respond child of the original message line 11.

Measurements

After implementation, the Fibonacci test application was used to measure execution time and delivery steps. The graph in Figure 6.3 shows the measured values for the calculation of Fibonacci numbers with different indices. Each value if the average of ten consecutive runs.

The graph shows that the number of invocations of the `deliver()` method is the main reason for the increased execution time since the values directly correspond with each other. Thus the performance can be improved by optimizing the binding algorithm for example by caching resolved cells. An approach to improve performance by decreasing delivery steps needed for the resolution of parent references led to an performance increase of only five percent.

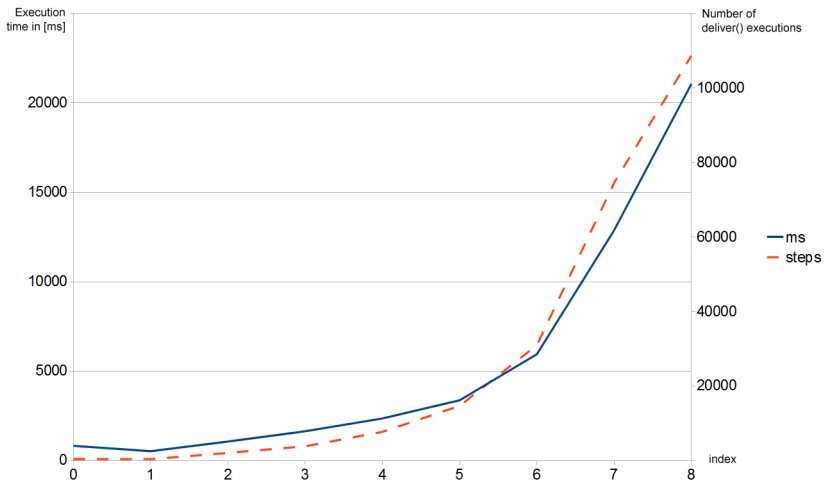


Figure 6.3: Measured execution time and invocation of `deliver()` for calculations of Fibonacci numbers with different indices.

Chapter 7

Outlook & Conclusions

7.1 High Level Language

The examples in Sections 3.2 and 6.3 show how the platform can be programmed by only using basic messages. To create applications more efficiently, a higher level programming language is needed. This section presents the draft of a language which uses as little syntactical constructs as possible to reach the expressiveness of modern programming languages. The constructs are described informally using examples.

Listing 7.1 gives an example of the syntax by implementing the recursive algorithm to calculate a number of the Fibonacci sequence as described in Section 6.3 using the high level language. The content of the listing corresponds to the definition of a reaction to create and define all cells of the application. This reaction would only need to be executed once since all created cells are always persistent.

Listing 7.1: Implementation the Fibonacci algorithm using a high level language

```
1  Fibonacci : ∘.Cell {
2    index : message .
3    [[index.equals 0].or [index.equals 1]] (
4      ifTrue [{index.respond index}] .
5      ifFalse [{
6        index.respond [[Fibonacci [index.subtract 1]].add
7            [Fibonacci [index.subtract 2]]]
8      }]
9    )
10 }
```

7.1.1 Extended Scope

As described in Section 3.3.3, the lexical scope of a cell does not include enclosing cells. For convenience, the parser of the presented language allows omitting preceding parent references which are added when the code is compiled into mailings. Its usage is therefore restricted to children of enclosing cells which already exist or are defined in the same compilation unit when compiling. As in Listing 7.1, message aliases can be specialized to resolve ambiguous references when omitting parent.

7.1.2 Answers

As described before, a mailing consists of the receiver and message cells. If a response is expected, a cell to which the response can be sent has to be provided as part of the message, due to the asynchronism of mailings (see Section 3.2.2). By convention, this cell is a child of the message with the name respond. Its default reaction creates a child response of the message cell with the received cell as its stem. As can be seen in the example in Section 3.2, this behaviour can be used for data flow synchronization by sending a message to response which will be repeated until response exists, i.e. a response was received.

The language draft provides a construct called *answer* which creates a container cell and accesses its response child implicitly. An answer is a mailing surrounded by brackets and can be used as a reference to the response of the container cell. Listing 7.2 shows an example of its usage and the corresponding list of equivalent mailings.

Listing 7.2: Implicit answers and equivalent mailings

```
1  message.respond [[True.or False].and [False.or True]]
2
3  cell.create.container1 ← ∘.False
4  True.or ← container1
5
6  cell.create.container2 ← ∘.True
7  False.or ← container2
8
9  cell.create.container3 ← container2.response
10 container1.response.and ← container3
11
12 message.respond ← container3.response
```

7.1.3 Definitions

To create and define a new cell with stem path and reaction requires a large number of regular mailings. Although the easiest case, creating a cell without reaction and a

static stem cell, can be expressed with a single mailing as in the example in Section 3.2.2. For more complex cases, the language provides a construct which allows compact cell creation and definition. Table 7.1 contains example usages of the cell definitions syntax and an explanation of their meanings.

Table 7.1: Examples usage of cell definitions.

Definition	Explanation

`A.NewCell : Its.StemCell {`
`  First.Receiver ItsMessage`
`  SecondReceiver AnotherMessage`
`}`
	Creates a cell NewCell as child of A and sets Its.StemCell as its stem cell (relative to NewCell). Defines NewCell's reaction with two mailings.

`[List {Its Reaction}].add SomeCell`	Creates an anonymous cell whose stem cell is List and adds SomeCell as an element. No further elements can be added since anonymous cells can only be referenced using their definition.
`True.or [°.False]`	Sends a new anonymous sub-cell of False to True.or.
`{do something} Cell`	Creates an anonymous cell with only a reaction and sends Cell to it to execute the reaction.
`True.ifTrue [{do conditionally}]`	Creates an anonymous cell with only a reaction and sends it to True.ifTrue. As a result, ifTrue sends a message to the anonymous cell and its reaction is executed.

7.1.4 Spaces

Many messages are often sent to the same cell or children of a common parent. To avoid repetition of a cell path in such a case, parentheses can be used to create a *space* in which all paths are relative to the preceding cell, which is especially useful in combination with anonymous cells. Table 7.2 contains examples of how to use this construct and their explanations.

7.2 Future Work

The current version of the presented software platform only covers the most basic features to prove the implementability of its programming model. This section describes

Table 7.2: Examples usage of spaces.

Usage	Explanation

`aList.add [Person (`
`  name : °.Literal.String.John`
`  age : °.Literal.Number.26`
`)]`
	Creates an anonymous sub-cell of Person with two children name and age and adds a new element to aList with the anonymous cell as its stem cell.

`[Point (x:x1. y:y1)].moveTo` `  [(x:x2. y:y2)]`	Sends an anonymous cell with two children to the moveTo child of a sub-cell of Point. This example illustrates how spaces can be used to simulate arguments in a method call.

some of the next steps and current approaches which lead to a usable personal computing platform.

Security For a distributed system, access control is a crucial feature. To prevent unwanted messages from being received, but also to provide encapsulation, fine-grained access control on object level is required. How this can be achieved efficiently will be the subject of future investigation. A possible approach is to use certificate-based access control [36] using white-/black-lists or roles [37]. The platform's architecture would lend itself naturally to serve as a distributed public key infrastructure.

The most critical security factor however is the end user [38]. In order to have a safe system, its users need to understand and be aware of security and privacy concerns. This requires educational work independent of any system but also clarity and transparency of the software platform.

Parallelism In order to prevent timing errors, the system needs to be extended by the ability to restrict the concurrent execution of reactions. Cells restricted in this way are not able to receive messages while their reaction is executing and therefore use a mailbox to store these messages. The restriction also has implications on the possible distribution of such cells which have to be analysed.

Performance Due to its complete distribution which results in having to resolve the receiver of every single mailing recursively, the performance of the prototype implementation is not as good as with existing commercial programming languages. Since it was not a requirement of the prototype, few experiments were conducted to quantify the performance and no measures were taken to improve it. Caching on multiple levels is considered as the most effective approach to improve performance. Results of name resolutions could be stored and re-used, as well as

memory location for cells in the same address-space. Since cells are principally mobile, communication partners can be migrated to other hosts to reduce network traffic and resulting latency. Mechanisms to ensure consistency have to be provided and their overhead considered to calculate the net performance increase.

Memory Management The presented programming model introduces new challenges regarding memory management which includes identifying and deleting disposable cells. Conventional garbage collection is not applicable since a cell never becomes unreachable due to the fact that all references are completely late-bound. Therefore, a more general approach is required that is not only used for application cells but also to user cells.

In a distributed system, memory management also includes the migration and distribution of entities. This requires the development of strategies which consider processor load, memory usage and network capacities. Ideally, these mechanisms work completely automatically but the ability for assisted manual optimization should be offered as well.

Library In order to develop portable systems efficiently, the software platform has to provide a standard cell library which provides reflection, hardware access and generic implementations for frequently needed functionality (as provided by [39]). The library can be distributed as well and does not have to reside in the local host, except for the parts of it that are implemented on the kernel level. It is also possible to replace any cell seamlessly with a native implementation for optimization. A key principle of the library's design has to keep meaning separated from optimization as suggested by [11].

Hardware access includes interaction with users and external devices. Graphical output uses real-world units instead of pixels and uses exclusively vector graphics (as done in [40]) to be independent of screen size and resolution. The library also contains cells for literal values, such as strings and numbers, cells for numerical and boolean algebra as well as collections and control structures. Any user can extend the standard library by creating a sub-cell of it.

Language The platform does not include a native language. Its native instructions are mailings consisting of receiver and message cells. To develop systems efficiently, a more expressive language is required. In Section 7.1 the syntax of a language draft was described informally which extends the basic instructions very little and thus can easily be translated into mailings.

The platform architecture should facilitate and encourage the use of different, possibly domain-specific, front-end languages. Because of a common underlying programming model, a system written in any language would be able to communicate seamlessly with any system on any host written in any other language. To increase end-user literacy, a graphical scripting-like way of programming should also be included.

Typing The programming model does not have a type system. Because of its value to system analysis and resulting coding assistance, a type system can be added as hints when needed (similar to [41]). These hints could be formulated as cell definitions expressing expected stem cells or children of messages and responses. These definitions would not have any effect on the system but could be used by an automatic analyzer to detect errors and provide coding assistance such as auto-completion.

Formalization To study the implications of distribution and concurrency more thoroughly, the semantics of the programming model will be formalized. Using model checking [42], the model can then be tested for exclusion of undesired situations such as dead-locks and inconsistency to prove its correctness [43]. Another advantage of a formalization is to translate it automatically into native code for different machines and thus produce completely compatible executables for multiple platforms.

Versions A desirable feature of the software platform and important for usability is fine-grained version control. All changes to any cell should be saved and be reversible. The system should allow the user to tag, discard and reverse to certain versions and also to create branches. Since this capability would not only apply to data but also to its representation, users could browse through any program like websites, always being able to return to previous views or keep a certain view open and branch into new views simultaneously, like opening new tabs in browsers [44].

It is not clear how to determine the scope of versions. The user should be able to control whether to reverse changes made only to a certain cell, to the cell and its children, or all changes to every cell. The last option would require performing all actions in a sandbox which can be committed or discarded by the user (similar to [45]). This would allow the user to conduct experiments with any system in a safe environment with complete control over side-effects.

Extensibility The goal of the platform is to be completely open and extensible which entails every part of the platform being exposed to the user by reflection. This is already the case for all parts except for the semantics of message sending. Since it is the only operation of the programming model, being able to modify its semantics would allow the user to adapt the semantics of the entire model if needed [46]. Design decisions like single inheritance or lexical scope could be changed for any part of a cell system while running, leading to a completely extensible software platform which is regarded as a crucial feature for its survival.

7.3 Conclusions

The thesis intends to help making the real computer revolution happen by laying the groundwork of a new software platform that encourages collaborative model building. First, the programming model was developed and implemented during several iterations using modern software development methods. Following this, the kernel implementation was connected with the local and remote systems. Finally, an object library and graphical development tools were implemented to support the development of example application which were used to test the system.

The result is a working prototype implementation of the software platform including two sample applications and an extensive set of unit tests. Although its low performance and immaturity prevent the platform from being practical it is a usable base for further investigation, experiments and improvements.

In conclusion, the thesis shows that it is possible to implement a completely abstracted and simplistic programming model in a concurrent and distributed software platform. A distributed name directory for path resolution could be implemented within the models structure. The model was constructed using a single kind of entity that represents an abstracted computing unit without the use of any low level concepts such as variable or assignments. It also has been demonstrated that message passing combined with hierarchical object identifiers is a suitable primitive operation that could form part of a common inter-platform language.

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Listings

Glossary

FDD Feature Driven Development, page 8

HTML Hyptertext Mark-up Language, page 3

HTTP Hypertext Transport Protocol, page 3

IEEE Institute of Electrical and Electronics Engineers, page 7

IP Internet Protocol, page 37

NATO North Atlantic Treaty Organization, page 7

TCP Transmission Control Protocol, page 37

TDD Test Driven Development, page 9

XML Extensible Mark-up Language, page 34

XP Extreme Programming, page 8

References

[1] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. iii, 49

[2] Alan C. Kay. **The computer revolution hasn't happened yet (keynote session)**. In *Proceedings of the eighth ACM international conference on Multimedia*, MULTIMEDIA '00, pages 1–, New York, NY, USA, 2000. ACM. 1

[3] Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. **Computing as a Discipline**. *Computer*, **22**:63–70, February 1989. Chairman-Denning, Peter J. 1

[4] Timothy Colburn and Gary Shute. **Abstraction in Computer Science**. *Minds Mach.*, **17**:169–184, July 2007. 1

[5] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. 1

[6] Roger Clarke. **A contingency approach to the application software generations**. *SIGMIS Database*, **22**:23–34, June 1991. 2

[7] Bertrand Meyer. **The Power of Abstraction, Reuse, and Simplicity: An Object-Oriented Library for Event-Driven Design**. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, **2635** of *Lecture Notes in Computer Science*, pages 236–271. Springer Berlin / Heidelberg, 2004. 2

[8] H. A. Ramadhan N.S. Kutti, Z.A. Al-Khanjari and J. Fiaidhi. **A Note towards Reshaping Java's Features**. *Journal of Computer Sciences*, **1**:450–453, 2005. 2

[9] Harold Thimbleby. **A critique of Java**. *Softw. Pract. Exper.*, **29**:457–478, April 1999. 2

[10] Ian Joyner. **C++?? - A Critique of C++**, 1992. 2

[11] K. Rose D. Ingalls D. Amelang T. Kaehler Y. Ohshima H. Samimi C. Thacker S. Wallace A. Warth A. Kay, I. Piumarta and T. Yamamiya. **STEPS Toward The Reinvention of Programming, 2008 Progress Report Submitted to the National Science Foundation**, 2008. 2, 61

[12] Frederick P. Brooks, Jr. **No Silver Bullet: Essence and Accidents of Software Engineering**. *Computer*, **20**:10–19, April 1987. 3

[13] **IEEE Standard Glossary of Software Engineering Terminology**. Technical report, 1990. 7

[14] Helmut Balzert. *Lehrbuch der Softwaretechnik, Teil 2: Softwaremanagement, Software-Qualitaetssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998. 7

[15] P. Naur and B. Randell. **Software Engineering, Report of a conference sponsored by the NATO Science Committee. NATO, 1968. Held at Garmisch, Germany**. *7th-11th October*, 1968. 7

[16] Edsger W. Dijkstra. **The humble programmer**. *Commun. ACM*, **15**:859–866, October 1972. 7

[17] W. W. Royce. **Managing the development of large software systems: concepts and techniques**. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. 8

[18] C. Larman and V.R. Basili. **Iterative and Incremental Development: A Brief History**. *IEEE Computer*, **36**(6):47–56, 2003. 8

[19] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. **Manifesto for Agile Software Development**. 2001. 8

[20] Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 10

[21] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999. 10

[22] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004. 10

[23] Alan C. Kay. **History of programming languages—II**. chapter The early history of Smalltalk, pages 511–598. ACM, New York, NY, USA, 1996. 13, 49

[24] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000. 49

[25] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996. 49

[26] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. **Modules as Objects in Newspeak**. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming*, **6183** of *Lecture Notes in Computer Science*, pages 405–428. Springer Berlin / Heidelberg, 2010. 49

[27] Gilad Bracha. **The Newspeak programming language specification, version 0.05**. `http://bracha.org/newspeak-spec.pdf`, 2009. 49

[28] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. 49

[29] Carl Hewitt, Peter Bishop, and Richard Steiger. **A universal modular ACTOR formalism for artificial intelligence**. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 49

[30] D. G. Kafura and K. H. Lee. **Inheritance in actor based concurrent object-oriented languages**. *Comput. J.*, **32**:297–304, July 1989. 49

[31] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, http://citeseer.ist.psu.edu/393979.html, 1996. 49

[32] Joe Armstrong. **A History of Erlang**. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM. 49

[33] H. Lieberman. **Thinking About Lots of Things at Once Without Getting Confused – Parallelism in Act 1**. MIT AI memo 626, May 1981. 50

[34] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004. 50

[35] David Ungar and Randall B. Smith. **Self: The power of simplicity**. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. 50

[36] Mary Thompson, William Johnston, Srilekha Mudumbai, Gary Hoo, Keith Jackson, and Abdelilah Essiari. **Certificate-based access control for widely distributed resources**. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, pages 17–17, Berkeley, CA, USA, 1999. USENIX Association. 60

[37] Amir Herzberg, Yosi Mass, Joris Michaeli, Yiftach Ravid, and Dalit Naor. **Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers**. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 2–, Washington, DC, USA, 2000. IEEE Computer Society. 60

[38] Mikko T. Siponen. **A conceptual foundation for organizational information security awareness**. *Inf. Manag. Comput. Security*, **8**(1):31–41, 2000. 60

[39] **GNU Smalltalk Library Reference**. `http://www.gnu.org`. 61

[40] K. Rose D. Ingalls D. Amelang T. Kaehler Y. Ohshima C. Thacker S. Wallace A. Warth A. Kay, I. Piumarta and T. Yamamiya. **Steps Toward The Reinvention of Programming**, 2007. 61

[41] G. Bracha. **Pluggable Type Systems**. In *Workshop on Revival of Dynamic Languages*, OOPSLA '04, 2004. 62

[42] Edmund Clarke. **Model checking**. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, **1346** of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0058022. 62

[43] G. J. Holzmann. **The model checker SPIN**. *Software Engineering, IEEE Transactions on*, **23**(5):279–295, 1997. 62

[44] et al. Alan Kay. **STEPS Toward Expressive Programming Systems, 2010 Progress Report Submitted to the National Science Foundation**, 2010. 62

[45] Alessandro Warth and Alan Kay. **Worlds: Controlling the Scope of Side Effects**, 2010. 62

[46] Ian Piumarta and Alessandro Warth. **Open, Extensible Object Models**. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, **5146** of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin / Heidelberg, 2008. 62

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other German or foreign examination board.

The thesis work was conducted from August 2010 to February 2011 under the supervision of Javier Jaén at Universitat Politécnica de Valencia.

Valencia,