Modelling Code Mobility

Karin Wrang Bruun

LYNGBY 2003
EKSAMENSPROJEKT
NR. 26/03

IMM

Preface

This master thesis is the result of work carried out in the period October 2002 to March 2003 at the section of Computer Science and Engineering at the department of Informatics and Mathematical Modelling at the Technical University of Denmark (DTU).

The work has been supervised by assoc. prof. Hans Henrik Løvengreen, whom I would like to thank for feedback, comments, and helpful suggestions.

I would also like to thank Harry B. Petersen, Mikkel Hansen, Benjamin Rasmussen, and Parminder Bedi for good company and support while writing the thesis.

31 March 2003, Lyngby

Karin Wrang Bruun

Abstract

Many of today's distributed systems rely on the movement of data, code, or even processes between the parts of the system. Among the technologies that support these forms of mobility are Java and several agent technologies.

This thesis will describe the construction of a new modelling language, the Mobility Language, which can be used to model systems with mobility and particularly all the mobility mechanisms used in Java. The work will also involve identification of a set of requirements, which should be fulfilled by any language that should be able to model all aspects of mobile code systems.

The distributed join-calculus is taken as the basis of the new language and it is modified and extended in order to allow code reuse and modelling of different mobility mechanisms. As part of this, the concepts of by-type bindings and component instances are introduced in the language.

The semantics of the language will be described in terms of a chemical abstract machine.

Keywords: Modelling languages, Formal semantics, Code mobility, Mobile agent technologies, Java, Chemical abstract machine (CHAM).

Contents

Chapter 1

Introduction

Many of today's information systems are distributed systems where several processes run concurrently and communicate over a network. Such distributed systems have evolved heavily during the last decades and systems are no longer static systems with static communication paths but they allow dynamic adding of new self-configuring components. Also, migrating code has become standard, e.g. in the form of Java applets or server proxies that are downloaded from a server and executed on a separate machine. Agent technologies take this one step further and allow running processes to decide by themselves when and where to move.

While all of this have become possible with today's technology, it has been hard for specification languages to keep up with this development. Many formal languages lack the ability to express dynamic process creation and movement of code and processes. However, there also exist a range of formal languages that have been designed to model mobility, but these languages are not always easy to use and cannot always model the necessary mobility concepts.

The formal languages that can express mobility take different approaches to how systems with mobility should be modelled. Where the more direct interpretation of the word "mobility" is that physical or logical components move and change position, it can also be understood as being the communication links between the components that move. The last approach is used in the actor model and the π-calculus, which are formalisms that allow communication links to be created dynamically and sent between processes. This mobility of communication links allows one to model that the processes themselves move and in this connection changes neighbourhood and thus communication links.

Other formal languages can express mobility more directly by having an explicit concept of locations between which processes or components can move. Such languages are the ambient calculus and the distributed join-calculus.

1.1 Thesis Objective

It is the aim of this project to work towards the construction of a practical modelling language with the ability to describe mobility, particularly the mobility mechanisms used in Java.

The modelling language should be easy to use and should therefore provide constructs that rather directly can model different mobility concepts. Well-known programming paradigms such as object-oriented programming and component based design should be supported. Also, the language shouldn't be too far from well-known programming languages.

Ideally the modelling language should provide constructs that allow the modelling of all the dynamic and mobile aspects of existing technologies. It should also be possible to model the more 'classic' concurrency mechanisms, like asynchronous message passing, remote procedure calls, rendezvous, and synchronization of processes. However, the main objective will be to construct a language that can model the mobility mechanisms used in Java and RMI.

The language should be provided with a formal semantics. This is necessary in order for a model to have a clear and well-defined meaning. A formal semantics will also allow the construction of tools that can aid the validation of a model, e.g. an application that can simulate the behaviour of a model. The next step would be to define a relation that allows one to reason about whether a model satisfy certain requirements, e.g. whether the model exhibit a certain behaviour. The construction of such a relation or the construction of other validation tools will however be outside the scope of this thesis. This project will only be concerned with the construction of a well-defined language that can model mobility.

When constructing the language, a lot of work will be put into deciding what exactly it is that the language should be able to express. This will be achieved through a study of existing technologies with mobility. Existing formal languages that can express mobility will also be studied in order to determine how the different mobility mechanisms can be modelled. The new language will thus build on ideas from the existing languages but these ideas will be extended or modified in order to construct a language that easily can express the required mobility mechanisms.

1.2 Thesis Overview

The following chapters will be included in the report:

Chapter 2 will describe some of the existing technologies which make use of mobile code or which can be used to build systems with mobile code or processes. Particularly some of the Java based technologies will be described.

Chapter 3 will analyze the described technologies and will extract a set of requirements. These requirements should be fulfilled by any modelling language which should be able to model the mobility mechanisms and all the other concepts used in the technologies.

Chapter 4 will describe some of the existing formal languages that can model mobility. The languages that will be described are the π-calculus, Nomadic π, the distributed join-calculus, formal versions of the actor model, and the ambient calculus.

Chapter 5 will analyse the described formal languages with respect to the posed requirements in order to identify the requirements each language fulfils and the language constructs that allow it to do so.

Chapter 6 will be concerned with the construction of a new modelling language, which will be named the Mobility Language. The chapter will take basis in one of the described languages and will step by step extend the language with constructs that allow further requirements to be satisfied.

Chapter 7 will formally describe the proposed Mobility Language and will define syntax, semantics and auxiliary functions.

Chapter 8 will give examples demonstrating the use of the Mobility Language. Some of the mobility mechanisms used in the described technologies will be modelled.

Chapter 9 will evaluate the proposed Mobility Language with respect to the posed requirements. A couple of extensions to the language, other approaches to the construction of the language, and future work will be suggested.

Chapter 10 concludes the project.

The reader should be familiar with the basic ideas behind software design and formal languages. Familiarity with parallel programming and distributed systems is expected, but all the concepts that concern mobility will be explained. Experience with Java and object oriented programming will also be an advantage. For the more theoretical part, a basic knowledge of structural operational semantics will be needed.

Chapter 2

Technologies with Mobility

A lot of technologies already exist that somehow explore the concept of mobility. Some systems allow the movement of passive code, others allow the movement of running processes. As the first step in the modelling of systems with mobile code, it is necessary to understand what it is these systems can do and thus what it is that should be modelled. Preferably, it should be possible to model and formalise the concepts that are already being used in existing technologies. This chapter will look into some of the existing technologies and describe how they work and how they can be used.

The motivation for introducing code mobility into systems will first be shortly described. The sections 2.2 and 2.3 will then look at the code mobility in Java and the way it is used in Jini to build open systems. The concept of mobile agents will be introduced and two particular agent languages will be described in section 2.4.

2.1 Advantages of Mobile Code

The ability to send code between different locations can give some advantages over traditional systems. Code mobility allows a computation to move to the location where the needed resources are. Without the code mobility, the computation would have to access the resource across the network.

Consider the example in figure 2.1. A client needs to perform a computation to which it needs a lot of data from a remotely located server. The client can either fetch all the data across the network by using a lot of remote procedure calls (RPCs). Or the client can send some code that knows how to perform the computation, across the network to the server. The code executes at the server and fetches all the required data using local procedure calls, performs the computation and sends the final result back to the client.

The last approach can have several advantages over the first approach:

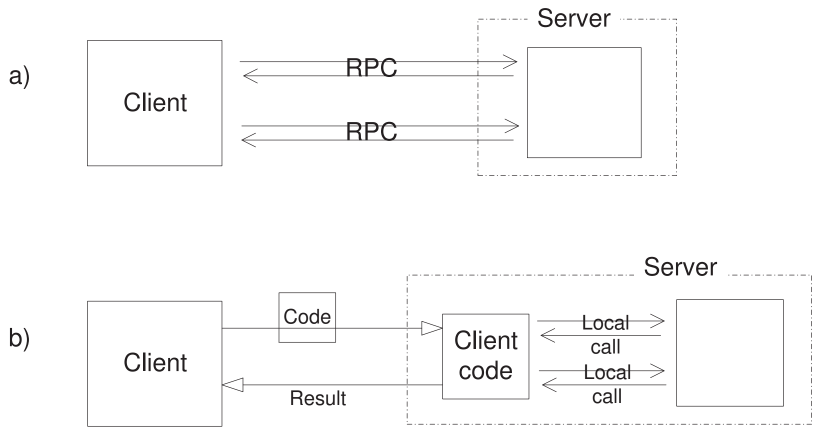

Figure 2.1: A client performing a computation needs some data located at a remote server. The client can proceed in two ways: a) The client uses remote procedure calls to obtain the data and then processes the data. b) The client sends some code to a remote server. The code executes at the remote server and uses local calls to obtain the needed data. The code processes the data at the server side and returns a single result to the client. If all the intermediary data can be compressed to a final result with much less data, this approach will result in much less network traffic.

Improved performance : Interaction among components in the same location is generally less expensive than interaction among components in different locations. Thus, the local procedure calls will be faster than the remote procedure calls.
Also, the approach can give less network traffic. If some form of semantic compression of the data can be made, such that the result returned to the client is only a small amount of data compared to the amount of data needed in the computation, the overall network traffic will be reduced.

Disconnected service : Once the agent is sent, the connection from the client can be shut down and the computation can still continue. The client can later connect to the network to receive a result.

Customisation : The server can have its functionality extended, without permanently affecting the size or complexity of the server and without one having to temporarily shut down the server.

Improved fault tolerance : With traditional remote procedure calls, the state of the computation is distributed across the network. This makes computation vulnerable to failures in the network and the components and it gets very hard to recover from failures. When the code travels to the server, the state of the computation is local and therefore less vulnerable to failures.

2.2 Java

Java is a technology that has driven a lot of development in mobile code systems. Not every Java programmer is aware of the mechanisms in Java that allow code mobility, but Java actually supports the movement of code through its class loading mechanism and its ability to serialize objects [Sun99][Sun][Som01]. The code mobility is used in the well-known concept of applets and it is a main factor in the working of remote method invocation (RMI), which will be described in the following.

2.2.1 Dynamic Class Downloading

The Java platform has the ability to dynamically download and run Java code. This is known from applets, where a browser starts a Java virtual machine(JVM), which downloads and runs the bytecode for the classes used by the applet.

A *class loader* handles the downloading of class files. When a JVM encounters a reference to a class not yet loaded into the VM, it asks the class loader to load the class. The class loader has a specific place it searches for the class files: the place indicated by the class loaders *codebase*. The usual bootstrap class loader has a codebase that points to the local Java Core class library and the directories given by the `CLASSPATH` environment variable. However, new class loaders can be created and given another codebase, and this allows new places to be searched for classes.

The class loaders in a JVM form a hierarchy, where every class loader has a parent class loader. The root class loader is the bootstrap class loader. When a class loader is asked to load a class, it first asks its parent class loader, which again asks its parent and so forth until the bootstrap class loader is asked. If the bootstrap class loader can find the class, it loads the class; otherwise the search for the class is delegated back down the chain of class loaders. Only if none of the parents of the original class loader can find the class, will the class loader itself try to find and load the class from its specified codebase.

In the case of applets, the JVM creates a new class loader with the codebase set to the URL (Uniform Resource Locator) that is specified as the applet's codebase. When a class needs to be loaded, the class loader will first ask its parents to find the class, i.e. the class will first be looked for locally. If none of the parents can find the class, the class loader will look for the class at the specified URL. The class loader can handle URLs of the forms `file:///`, `ftp://` and `http://`, but in the case of applets it will usually be from an HTTP server that the classes should be fetched.

This mechanism in Java allows a system to run a program, which has never been installed on the disk; the program is simply downloaded when needed.

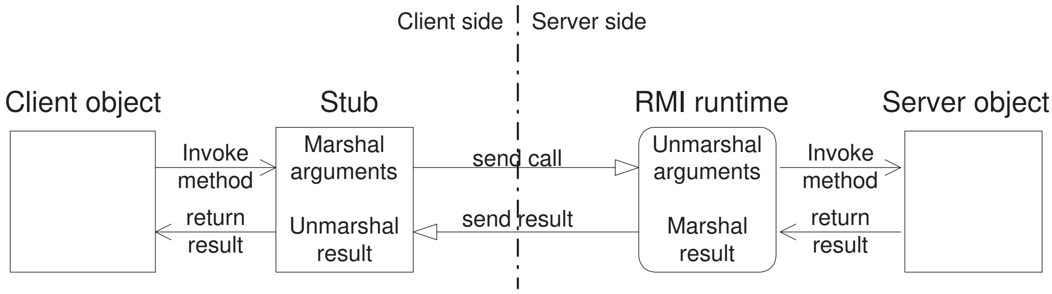

Figure 2.2: The client and the server object interact by means of a stub on the client side and the RMI runtime system on the server side.

2.2.2 Remote Method Invocation

RMI is Java's version of RPC extended to distributed objects. RMI allows objects in different JVMs to call each other's methods. RMI builds on the client-server paradigm, where *server objects*[1] offer operations that can be used by *client objects*.

Client and server objects don't interact directly but by means of *stubs*. All classes that should act as servers have a corresponding stub class that are generated by the `rmic` compiler. The stub class will have the same interface as the server class and it will know how to contact the server object across the network, using the appropriate network address, port number, etc.

For a client and a server object to interact, the client should obtain a corresponding stub object through which all communication will go. See figure 2.2. The client can make calls to the stub object, just as it would to any other local object. The stub will then *marshal* the arguments (translate from application format to transport format) and forward the call across the network to the remote host. The RMI runtime system on the receiver side will know all the server objects offered on the machine. It will unmarshal the arguments and invoke the method on the server object. A result will be communicated back through the same intermediaries.

For this to work, the client must be able to obtain a stub object corresponding to the server object. Java's *serialization* mechanism allows an object to be serialized into a stream of data, from where the object's state can be reconstructed at a later time. This stream can also be sent across the network to a remote JVM. This allows stub objects to be communicated to client objects.

The distribution of stub objects goes through the *RMIregistry*. This registry is a server object running on every host that offers server objects. The registry's stub

[1] I will here use the term "server object" to mean an object that extends the class `java.rmi.server.RemoteServer`. In the literature such an object is often referred to as a remote object. However, I find that term rather confusing, as the object not necessarily is located remotely.

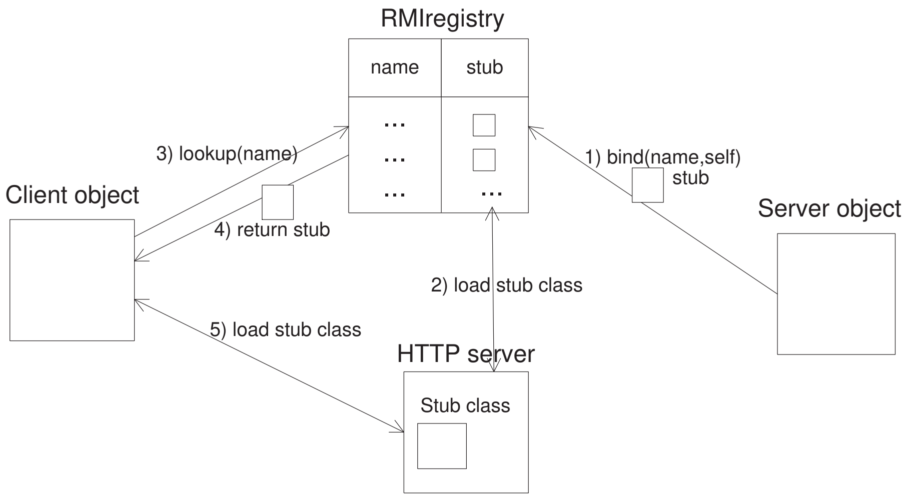

Figure 2.3: The stubs of server objects are distributed through the RMIregistry. 1) A server object registers with the RMIregistry which results in its stub object being uploaded to the registry. 2) If the JVM of the registry doesn't know the stub class, the class will be downloaded from a specified URL. 3) A client object, which wants to make use of the server object, asks the RMIregistry for the stub of the specific server object. 4) The stub is returned to the client object. 5) If the JVM of the client doesn't know the stub class, the class will be downloaded from a specified URL.

can automatically be obtained by every object and used to communicate with the registry.

When a server object wants to make its operations available, it registers with the RMIregistry under a chosen name. See arrow 1 in figure 2.3. This will result in a stub object being uploaded to the registry. A client object can lookup in the registry under the chosen name (arrow 3) and receive a stub to the server object (arrow 4).

However, even though a client can obtain the data for a stub object, it may not know the class of the stub object. This class code should somehow be made available to the JVMs, e.g. from an HTTP server, and the JVM should be told where to find the class files.

Therefore, when the stub object is sent as argument in an RMI call, the stub that marshals the stub object will make sure that the stream gets annotated with a codebase telling where the class code can be downloaded from. The JVM that receives the stub object, will create a class loader with the codebase set to the codebase of the received object. If the object's class isn't available locally, it will be fetched from this place (arrow 2 and 5 in figure 2.3).

The serialization and codebase annotation aren't limited to the stub objects. All objects passed as parameters to or returned from an RMI call, will be serialized and annotated with a codebase. However there is a single exception: When a server object is marshalled, it will be replaced with its stub object. This is how the stub object gets registered with the RMIregistry and how it otherwise can be handed out to remote client objects.

2.3 Jini

Jini [Sun01a] [Sun01b] [Som01] is a Java based technology used for *federating* groups of devices and software components into a single, dynamic distributed system. It is an example of an *open system*: It allows the addition of new components, the replacement of existing components, deletion of components, and changes in interconnections between components. All this without disturbing the functioning of the system.

Jini is just one technology among many that provides an architecture for open systems with device interaction. Other examples are Web Services, OSGi, UPnP (Universal Plug'n Play) and Salutation [Cal99]. The systems work in different layers of the network. UPnP, for example, provides a set of protocols that allow devices to announce their presence and discover each other using multicast. UPnP also addresses the problem of assigning an IP address and a DNS name to new devices; an aspect that Jini doesn't touch upon at all.

2.3.1 Architecture

The components in a Jini federation can offer *services* to each other and use each other's services. A service is represented by a Java interface that defines the operations that can be used to access the service. Only the type of service, i.e. its interface, is of interest to the user, not how the service is implemented. The service itself may be implemented using several Java objects or hardware devices.

A user that needs a particular operation can find all the services in the federation offering that operation at that particular time. The user doesn't have to know the location or name of such a service in advance; it only needs to know the interface of the needed service. This is opposed to RMI, where a client needs to know the name and location of a server object.

In Jini, all communication between services goes through *proxy objects*, just as is the case in RMI where all interaction with server objects goes through stubs. Java's ability to dynamically download and run code is used to transport these proxy objects across the network to the users of the services. A proxy object has the interface of the offered service, and a user will make local calls to the proxy object. However, unlike RMI stubs, the proxy object can do more than just forward the call to the service implementation. The proxy may be just an RMI stub but it can also be an object that does some or all of the computation locally and maybe makes use of

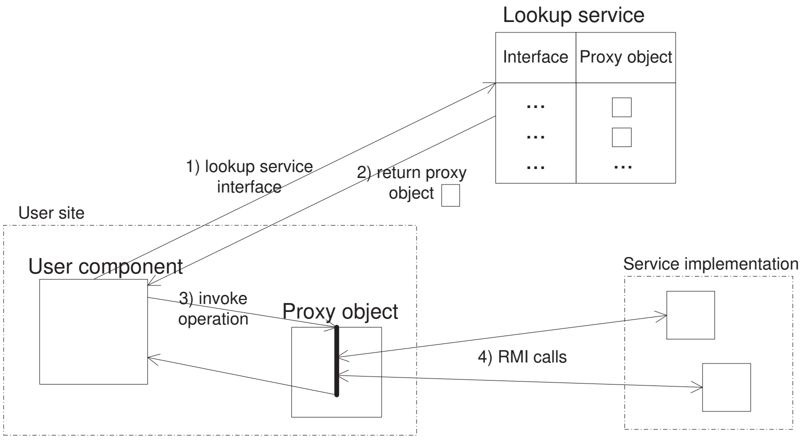

Figure 2.4: The use of a service goes through a proxy object. 1) The user component contacts a lookup service to get a proxy to the needed service. 2) A proxy object is returned to the user component. 3) The user can make local calls to the proxy object to use the operations offered by the service. 4) The proxy object can perform the computation locally or contact a remote service implementation using RMI calls.

several RMI calls to a remote service implementation. It may even use other services itself. See figure 2.4.

Addition of new services and discovery of services go through a *lookup service*. The lookup service is part of the Jini infrastructure and it is the central service in a Jini federation. When a component wants to join a federation, it starts by discovering a lookup service. This is done by multicasting a request on the local network for any lookup service to identify themselves. The component will then receive a proxy object to the lookup services in the network.

A component can offer a service by *registering* with the lookup service. It registers a proxy object under the offered interface name and optionally some attributes, to further identify the offered service. A component that needs to use a service, contacts the lookup service and queries on the interface name and the attributes. It will then obtain a proxy object for an appropriate service. See figure 2.4.

Jini extends the event model known from Java to the support of distributed events. Objects may register their interest in certain events in other objects, and receive notification when such events happen.

2.3.2 Handling of Partial Failure

In a distributed setting, components can fail, be removed or otherwise become unavailable. Jini registrations are therefore only valid for a certain time period, the *lease* period. Services must renew their registration before the lease expires, otherwise the registration will be removed from the lookup service. This is to ensure a correct picture of the components in the system.

Also access to a service is lease based, i.e. only granted for a certain period of time. This is used to figure out when to free a resource. If a lease is not renewed before it expires, both the user and the service provider will know that the resource can be freed. In this way a resource will always be freed, even though the client fails or the network connection is down.

Another concept that can make a system work correctly in the presence of partial failure, is that of *transactions*[FH00]. A user may need operations from several services in the system, but if one of these services fails, the user can't complete its computation and the state of the system may become inconsistent. Transactions allow operations to be grouped together, such that either all of the operations complete or none of them do. This will ensure that the state remains consistent.

Jini supports transactions through a transaction service that guides a set of participants (the operations involved in the transaction) through a two-phase commit protocol. In the first voting phase, all the operations involved in the transaction vote whether they have completed their computations. If all operations succeed, the transaction is committed and all operations complete. If a problem occurs, the transaction is aborted and all operations leave the state unchanged.

2.4 Mobile Agent Technologies

Though Java has the ability to download and run code that isn't already present in the JVM, the class downloading happens implicitly when needed. To make an object with corresponding class code move to a remote JVM, the object has to be wrapped in an RMI call to a server object in the remote JVM. Other systems allow one to move code more directly, particularly systems using *mobile agents*.

A mobile agent is a component that includes code and a running process. A mobile agent can move between locations in the network, and on arrival to a new location continue the execution of the code. The decision of when and where to migrate is usually taken by the agent itself.

Mobile agents are useful when a needed resource is located at a remote location, this may be processor time, a procedure, or data. A mobile agent can travel to the remote location and do the necessary computations while it is co-located with the resource.

Applications using mobile agents are location aware, i.e. the location where a component is executed is a first-class element in the application design. This is opposed to the location and migration transparency that is usually provided by middleware and operating systems.

All agent systems have as basis some kind of *agent run-time system* that should be present at every network node. The run-time system takes care of data marshalling, security, etc. when an agent migrates. However the run-time system does not have any control over when and whereto an agent should migrate.

Several agent technologies exist but they differ in how much of an agent's state is migrated with the agent. In some agent systems the agent is moved with all of its state including program pointer. In other systems the agent starts running from a particular place in its code every time it has been moved. In some agent systems, it is the job of the programmer to pack the necessary data space of a component before it is migrated, and then rebuild the execution state after the migration. In other agent systems, this is taken care of by the run-time system.

Two particular mobile agent technologies will be described in the following. The Telescript technology and the Java based aglet technology have been chosen because they handle the migration of an agent's state differently and they have different politics for which objects should migrate together with an agent. However many other agent technologies exist, see [FPV98] and [GKCR00].

2.4.1 Telescript Agents

The Telescript technology [Whi97] [Gen95] is one of the first commercial mobile agent systems. It is designed for the electronic marketplace, and it assumes that a Telescript *engine* that can host agents is present at the involved network nodes.

A system consists of *agents* and *places*, which are active objects acting on behalf of some *authority* (an individual or organization in the physical world). Each agent and place has a telename and a teleaddress. The teleaddress is the network address of the engine where the object is running. The telename consists of a unique name and the name of the object's authority.

Places are stationary active objects that offer some service on behalf of their authority. Several places can be located on the same Telescript engine and a place can occupy another place, i.e. the places can form a hierarchy. All places that run on an engine operated by a particular authority constitute a *region*. All the regions make up the Telesphere. See figure 2.5.

Agents are active objects that can travel between places in the Telesphere. Agents occupying the same place can *meet* and get a reference to each other so that they can call each other's procedures. For two agents to meet, both agents have to agree. Agents in different places can *connect* and communicate by exchanging messages. Agents can interact with the place they occupy by calling the procedures of the place, using the reference they got when they entered a place.

An agent travels from one place to another by executing a `go`-instruction, specifying the destination. The destination can be specified as either

- the one place with a given name

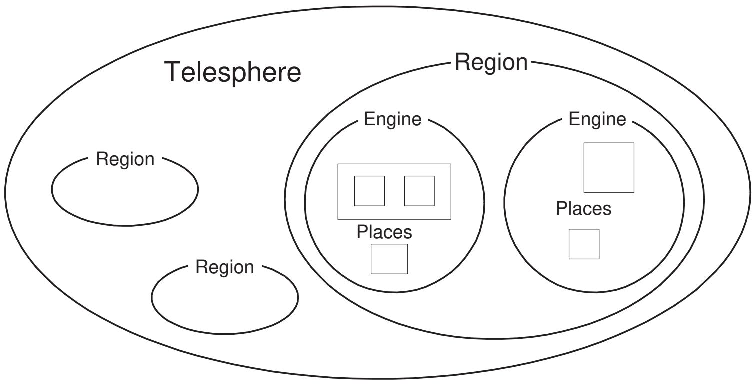

Figure 2.5: The hierarchic organization of places in the telesphere.

- any place of a given authority
- any place at a given teleaddress
- any place in a given region
- any place of a given class
- a combination of the above

The travel is initiated by the agent itself. An agent can also create a remote clone of itself, using a `send`-instruction.

Before an agent can enter a place, it has to be granted access by the place and the region. This will usually be based on the agent's authority.

An agent is given three *permits*; one from its authority, one from the place it is occupying and one from the region the place belongs to. The permits grant certain capabilities, e.g. to create other agents or to access certain resources. All three permits have to grant a capability before the agent can use that capability.

Every object has an *owner*, which is either an agent or a place. Every agent or place is owned by itself. Other objects are initially owned by the owner of the object that created the object. The ownership of an object may be handed over to another object. When an agent migrates, all the objects owned by it, migrate with it.

References to objects can be either *protected* or *unprotected*. A protected reference is a read-only reference, i.e. the object can't be modified through a protected reference. This can be used to give different agents different possibilities with an object. For example, an unprotected reference could be given to only those agents that have a certain authority.

The concept of authorities and permits is used to give a high degree of security in the system, such that places can be protected against malicious agents, and agents can be protected against malicious places.

2.4.2 Aglets

Aglets [Ven97] is an agent technology that builds on Java. An *aglet* is an active object that can travel between aglet *hosts* at different locations and carry out computations. An aglet can contain several active threads.

An aglet can create new aglets, it can dispatch itself or other aglets to a remote host, it can create a clone of itself or other aglets at a remote host, and it can retract other aglets from a remote host to the local host. An aglet is autonomous in the sense that it can decide by itself whether it wants to move or not, and where to. Other aglets can request that an aglet moves, but it is up to the aglet itself to decide whether it wants to do so.

Aglets interact through *proxy objects*. If an aglet wants to request another aglet to take certain actions, it invokes the corresponding method on a local proxy object. The proxy object then forwards the request to the aglet. This is just like RMI. Aglets can also communicate by message passing, both synchronous and asynchronous. This message passing also goes through the proxy object of the receiver. An aglet obtains a proxy object to another aglet when it creates the aglet. Every aglet has a unique identifier, and this can also be used to obtain a proxy object to an aglet.

When an aglet travels between hosts, it is actually the destination aglet host that uploads the class files and the state of the aglet. The object serialization mechanism of Java is used to code aglets and transport them between the different hosts. This means that not all of an aglets state travel with the agent. Execution stacks and registers aren't brought with the aglet, which means that temporary variables used in the computation are lost, and the aglet will have to start over from a particular point in the program because program counters are also lost.

Before an aglet is dispatched to another host, the current aglet host will call the aglet's `onDispatch()` method, which allows the aglet to decide whether it wants to move and if so to pack the necessary state information into data fields. When an aglet arrives at a new host, the host will invoke the aglet's `onArrival()` method, which allows the aglet to restore some state information or take whatever action is necessary. Then the aglet starts executing from the beginning of its `run()` method.

Because the Java serialization mechanism is used to transport an aglet, it will be this mechanism that decides which objects should migrate together with the aglet. The serialization mechanism will simply make a copy of all referenced objects and send these together with the aglet.

Chapter 3

Aspects of the Technologies

The technologies described in chapter 2 have many different properties and use different concepts in the construction of a system. This chapter will try to identify the different aspects of the technologies and will lead to the identification of a set of requirements to a modelling language for mobile code systems. These requirements will mostly focus on which concepts a language should be able to model.

The requirements suggested in this chapter are just wanted properties of a modelling language, needed if the language should be able to express every aspect of all of the technologies. However, there are also aspects of the described technologies that will not be touched upon here because the aspects are rather specialised and aren't necessary for the modelling of mobility. Reflection in objects and activation and deactivation of components will for example not be treated, though these concepts are part of the Java technology.

The sections in this chapter will each be concerned with one aspect of the technologies. The concrete terms and concepts of the technologies will be related to more general concepts whenever possible, and a set of requirements relating to each aspect will be identified. A full list of the requirements can be found in appendix A.

3.1 Component Types

One of the most fundamental aspects of the technologies is the kinds of components that they operate with. Generally, the technologies are all component based in the sense that a system will be divided into smaller entities (the components), which may interact with each other. The different technologies operate with different kinds of components.

3.1.1 Components in the Technologies

Java operates with classes and objects. An object is created from a class, which determines the behaviour of the object. The same class can be used to create several objects. Besides being connected with a class, an object further has some data attached.

When an object is sent across the network, the object's data will be sent together with a class descriptor telling the name and codebase of the class. The class itself is not sent together with the object but can be fetched separately. However, both the object data and the class definition will have to be present at the same JVM to make up a full executable object.

Objects are generally passive components that won't execute unless their operations are invoked. However, the creation of some objects can start a thread of execution, e.g. objects of the class Thread. The start of an application will also start a thread of execution. This thread of execution will drive the computation forward and make the objects call each other's operations. This is for example also the case in Jini, where the client components in a federation all have an attached execution thread that drives the computation and invokes operations in the needed services.

The agent technologies also operate with objects and they further have agents as a special kind of component. Every agent has an attached thread of execution that will follow the agent when it moves from location to location. In the Aglet technology, an aglet can even have more than one thread of execution. In the Telescript technology an agent will continue the execution where it left before the movement, whereas an aglet will have lost all information about its execution state before the movement.

To summarize, the technologies operate with several kinds of components and component parts: Classes, object data, class and object data combined to executable but passive objects, objects with an attached execution thread, and information about the state of an execution thread. A system is built from several components, and several execution threads can be running concurrently. New components can be created while a system is running.

3.1.2 General Component Types

A classification of components has been suggested in [FPV98]. Three kinds of components are identified: *code* components, *resource* components and *computational* components. Code components represent a know-how to perform a particular computation. Resource components are data or devices used during the computation. Computational components are active executors that can carry out a computation as specified by the corresponding code component.

A computation can only be carried out if both the code component, the computational component and the necessary resource components are present. Such a

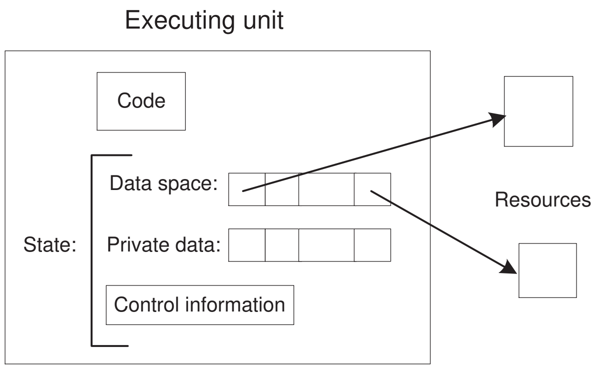

Figure 3.1: The structure of an executing unit. The executing unit has a code component and a state. The state consists of control information, private data, and references to external resources.

collection of a computational component, a code component and a state containing references to the needed resources is called an *executing unit*. See figure 3.1. The *state* is further split into the *data space* and the *execution state*, where the data space contains the references to external resources, and the execution state contains *private data* and *control information* for the executing unit. An executing unit represents sequential flow of computation (i.e. a single execution thread).

This suggested classification and structure of components has some similarities with the structure of the components in the described technologies. The code component corresponds to a class, the data space and private data of an executing unit correspond to an object's data, and the control information corresponds to the execution state. However, the grouping of these component parts into one single unit, the executing unit, doesn't seem appropriate to describe a situation where a single executing thread controls the execution of several objects. A slightly different classification will therefore be used.

Three kinds of components are identified: *code components*, *passive components* and *executing components*. A code component describes the behaviour of a passive component. A passive component is made up of a code component and a *state*. The state is split into *private data* and the *data space*, which consists of references to other components and external resources. An executing component consists of a passive component and *control information* for the computation, e.g. information about the next computation step or temporary variables. An executing component represents a single sequential computation. See figure 3.2.

Using this classification, a class is conceived as a code component, an object is conceived as a passive component, and Thread objects, Telescript agents, and aglets are conceived as executing components. The passive component of an executing component can have references to other components, passive as well as executing. This allows other passive components to be used in a computation, and it even allows

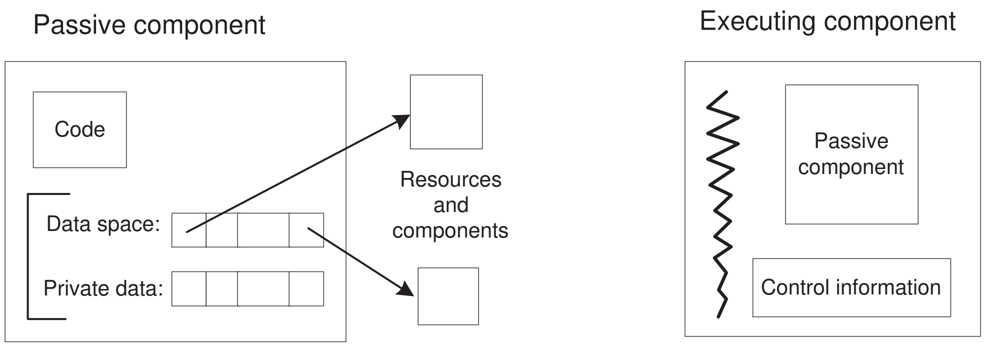

Figure 3.2: The structure of passive and executing components. A passive component has a code component and a state consisting of private data and references to external resources and other components. An executing component consists of a passive component and control information.

cooperation with other executing components.

The passive component in figure 3.2 is depicted with the code component inside the passive component. A code component can in fact be shared between several passive components, so a reference from a passive component to an external code component can seem more appropriate. However, the code component is an important part of a passive component and should be located together with the state in order for the passive component to be executable. The code component is therefore depicted inside the passive component.

3.1.3 Requirements Concerning Components

The component aspect of the technologies gives rise to the following requirements to a modelling language:

- It should be possible to model passive components consisting of a code component and a state.
- It should be possible to model executing components consisting of a passive component and control information.
- It should be possible to combine several components into a concurrently executing system.
- It should be possible to express dynamic creation of passive and executing components.
- It should be possible to use the same code component in several passive components.
- The language should reflect that a passive component can only be executed if both the code component and the state is present at the same location.

3.2 Locations

All the described technologies have some form of location concept.

Java objects are located at and executed by a JVM and an object may move from one JVM to another. Classes can also be loaded into a JVM from the file system or from a remote HTTP server. Aglets run at an aglet host and may move from host to host. Telescript agents are located at places and can move from place to place. The places themselves run at Telescript engines, which are divided into regions.

JVMs, HTTP servers, aglet hosts and Telescript places, engines and regions can be seen as *locations*. By a location is meant something that hosts resources and components and supports the execution of executing components. Components may move between locations but they cannot execute while they are moving. Locations can form a hierarchy where some locations are situated inside another location, and the locations may themselves move between these higher level locations.

The location concept is interesting because components and resources at the same location can interact more directly than components at different locations. System resources, such as files and hardware devices, can generally only be accessed by components co-located with the resource. If a file or a printer is needed by a component at another location, the request for the resource will have to go through a component co-located with the file or the printer. The request for a file at a file server will for example have to go through a server process that controls the access to the files.

Telescript agents can meet with other agents running at the same place and be allowed to call their methods. This isn't possible for agents running at different places. An agent can also get access to resources and procedures at the place it is occupying, but references to such resources and procedures will be "voided" the moment the agent leaves the place.

The location aspect gives rise to the following requirements to a modelling language:

- The language should have a way of modelling that some components and resources are co-located while others are at different locations.
- It should be possible to express a hierarchic structure in the locations.
- It should be possible to specify that interaction with certain components or resources only is possible from within the same location.

3.3 Mobility of Components

The most interesting aspect of the described technologies is their ability to move components, particularly code components and executing components, from one location to another. This changing of the location of a component exists in different forms in the different technologies.

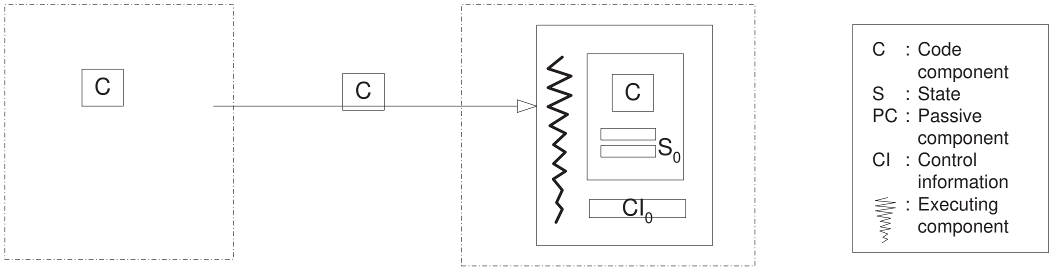

Figure 3.3: A code component can be moved from one location to another and become part of a new executing component. The code component can either be fetched by an executing component already at the second location (as is the case with applets) or it can be shipped by an executing component at the former location (as is the case with remote evaluation).

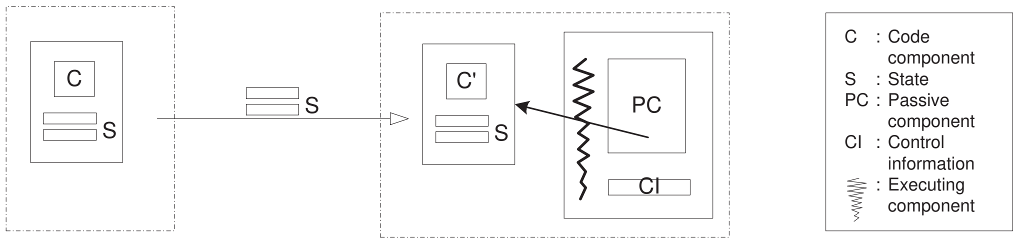

Figure 3.4: The state of a passive component can be moved from one location to another and become part of a new passive component. This is what happens in Java when an object is being moved. It can either be shipped or fetched.

3.3.1 Mobility in the Technologies

The mobility mechanisms in the described technologies will in the following be related to the kind of component that moves and what becomes of the component at the destination.

Applet classes can be fetched and executed. That is, a code component can be fetched from one location to another and used to start a new executing component (see figure 3.3).

The functioning of RMI relies on object data being sent from one location to another and corresponding class definitions being fetched from an HTTP server to the location. The resulting passive component will be used by other components already at the location. That is, the state of a passive component is sent from one location to another and forms a new passive component with a code component at that location (figure 3.4). If the code component isn't present at the location, it can be fetched from another location (figure 3.5). The resulting passive component will be used by an executing component already at the location.

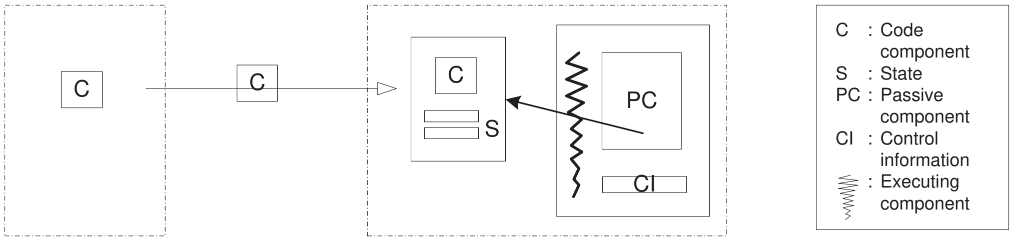

Figure 3.5: A code component can be moved from one location to another and become part of a passive component that will be used by an executing component. The code component can either be fetched by the executing component already at the second location (as is the case with class definitions in Java/RMI) or it can be shipped by an executing component at the former location (as is the case with remote evaluation).

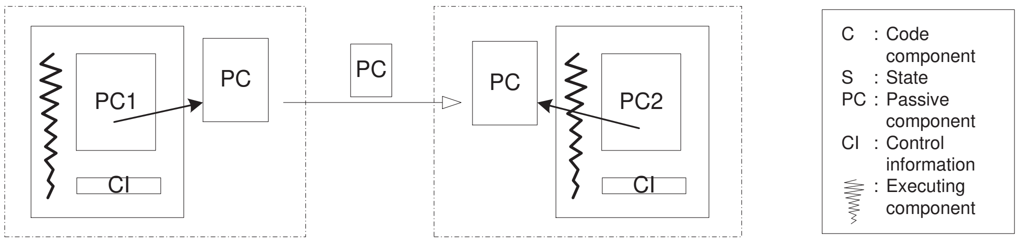

Figure 3.6: A passive component can be moved as a whole; from being part of the computations at one location to being part of the computations at another location. This is a more abstract view of object movement in Java.

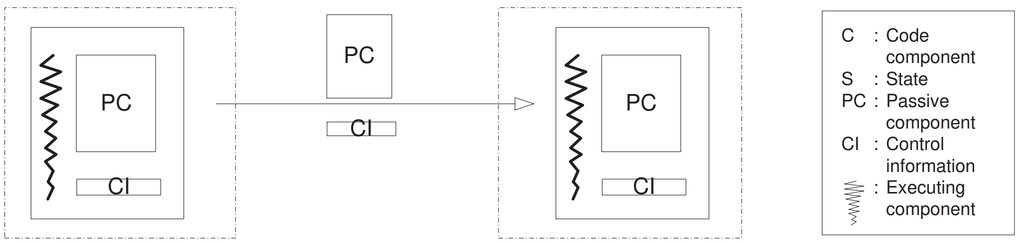

Figure 3.7: An executing component can move from one location to another, as is the case in Telescript. The executing component can either move itself or produce a remote clone of itself or it can be shipped or fetched by some other executing component.

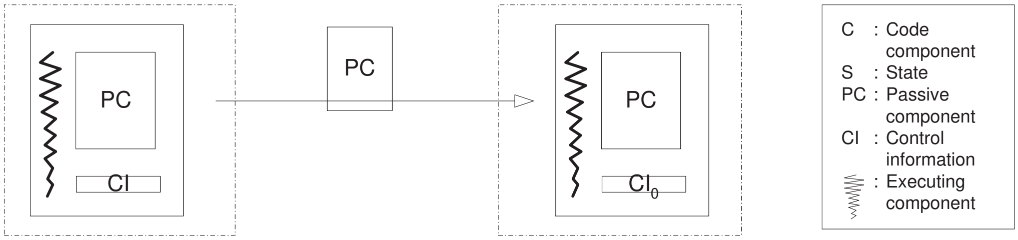

Figure 3.8: The passive component of an executing component can be moved from one location to another and become part of a new executing component. This is the case with aglets. The component can either be shipped, fetched or cloned remotely.

A user of RMI doesn't have to know that only the object data is being moved when an object is sent from one JVM to another, and that the class is assumed to be present at the destination. It will often be convenient to see the movement of an object as the movement of all of the passive component, i.e. both code and state (figure 3.6). The same view will be taken on agents in the agent technologies where the agent run-time system handles the movement of code and state separately.

Telescript agents can migrate from one place to another or create a clone of itself at another place. That is, the passive component and the control information of an executing component can move to another location and start a new executing component (figure 3.7).

Aglets can ship itself or other aglets to a remote aglet host, it can make a clone of itself or other aglets at a remote aglet host and it can fetch another aglet at a remote host. The aglets lose track of how far they have got in the execution when they are moved. That is, an executing component can be shipped to or fetched from a remote location but without its control information (figure 3.8).

Though it isn't possible in any of the technologies described here, some technologies (e.g. Obliq [FPV98]) use the concept of remote evaluation: A code component is shipped to a remote location and executed there, either in a new executing component (figure 3.3 where the code is being shipped instead of being fetched) or as part of an existing executing component (figure 3.5 where the code is being shipped instead of being fetched). The executing component that shipped the code may suspend until the execution of the code terminates and returns a result.

Other combinations of components that move to a new location and form other components can be imagined. However, I haven't come across any other combination in any existing technology.

When a component is being moved, it is in fact often a copy of the component that should be moved. When an HTTP server offers classes to be fetched, it will be copies of the classes that will be distributed. Likewise, when objects are communicated as part of an RMI call, it will be copies of the objects that are sent across the network.

In the agent technologies, remote clones of agents can be created, i.e. copies of agents can be sent across the network.

When a component (or copy of a component) is being moved, the destination location needs to be specified somehow. Using RMI, an object given as argument to an RMI call will simply be moved to the same location as the server object. The stub object that takes care of the marshalling will know the exact network address etc. but the client object will not need to know this. In the agent technologies, an agent will need to name the destination location more directly. A Telescript agent can name one particular place but it also has the possibility of specifying just any place that fulfils some requirements regarding functionality or authorities.

3.3.2 General Mobility Concepts

In [FPV98] the authors give names to the different kinds of component movements. They operate with two kinds of mobility: *strong* and *weak* mobility. It is strong mobility when an executing component can move to another location bringing all of its state information (the control information and the state of the passive component). Otherwise it is weak mobility. So of the described technologies only the Telescript technology provides strong mobility.

Strong mobility can take two forms: *migration* and *remote cloning*. Both kinds of mobility are provided by the Telescript technology. The movement of the executing component can either be initiated by the component itself or be triggered by some other executing component. In the first case it is called *proactive* migration, in the second case it is *reactive* migration.

Weak mobility can take several forms. A component can either be *fetched* from a remote location or be *shipped* to a remote location. The migrated component can in both cases either become part of an existing executing component, or it can be used to start a new executing component. In the first case the moving component is called a *code segment*; in the last case it is called *stand-alone code*.

The authors also distinguish between *asynchronous* and *synchronous* movement. In the last case, the component requesting the movement will suspend until the moved component has been executed. In the former case, the component requesting the movement will continue execution immediately.

Using these terms, Java's class loading mechanism and object serialization provides weak mobility with both fetching and shipping of code segments. The downloading of applets can be seen as fetching of stand-alone code. Aglets provide weak mobility with both fetching and shipping of stand-alone code. Systems using remote evaluation provide weak mobility with either synchronous or asynchronous shipping of either code-segments or stand-alone code.

3.3.3 Requirements Concerning Mobility

The mobility of components gives rise to the following requirements to a modelling language:

- It should be possible to model that a component or part of a component moves from one location to another location.
- It should be possible to model the movement of the following parts: a code component, the state of a passive component, all of a passive component and all of an executing component.
- It should be possible to model whether the moved part is a code segment that will become part of a passive component at the destination or whether it is stand-alone code that will become part of a new executing component.
- Components should have a way of naming which location itself or other components should move to. It should be possible to name both a particular location and a set of locations that fulfil some requirements.
- It should be possible to specify that a component migrates at its own will, that it is being shipped by some other executing component, and that it is being fetched by some other executing component.
- It should be possible to model that a copy of a component (passive or executing) is being moved.
- It should be possible to model both synchronous and asynchronous moves.

3.4 Component Bindings

A component may have references to external resources and other components (the data space). The question of what happens to the resource and component bindings upon migration is treated differently in the different technologies.

3.4.1 Binding Handling in the Technologies

When a Java object is sent as part of an RMI call, a copy of all the referenced objects will be sent as well. However, for server objects it will only be the corresponding stub that is sent, and this can be seen as the object getting a reference across the network to the server object. References to resources, e.g. files or hardware, will either become invalid or will be pointing to similar resources at the destination location, depending on the kind of resource.

Aglets are moved using the mechanisms of Java, so a copy of all the referenced objects will be moved together with the aglet. Other aglets are referenced through proxy objects, so only these proxy objects will move with the aglet, not the aglets themselves. A proxy object has a reference across the network to an aglet.

A Telescript agent may have references to several objects but upon migration it only brings the objects, which it owns. Objects owned by other agents or places will stay where they are and the references to these objects will be voided, i.e. the objects won't be referenced any more.

What happens upon migration to the references in a component's data space seems to depend on what it is that is referenced.

3.4.2 General Binding Concepts

In [FPV98] a classification is made of the different kinds of resources and of the different kinds of bindings to resources. Resources are divided into *transferable resources* (e.g. data) and *non-transferable resources* (e.g. a printer). Likewise, resource references are divided into three kinds: *binding by type*, *binding by value* and *binding by identifier*.

A resource is bound by type if it is only required that the bound resource has a certain type but it doesn't matter what the actual value or identity is. This kind of reference would typically be used for resources that are available at every location, e.g. the display, system variables or libraries. A resource is bound by value if the bound resource should not only have a certain type but also a certain value or content. This could be the case with replicated data, where the exact identity of the bound resource doesn't matter, but the content should be the same. Binding by identity is the strongest form of binding, where the bound resource at any time should be a uniquely identified resource.

How the bindings to referenced resources should be handled upon migration will depend on the kind of resource and the kind of binding.

If a component has a reference to a resource that is bound by type, there is no reason for the resource to migrate with the component. The reference should simply be rebound to a similar resource at the destination location.

If a component has a reference to a resource that is bound by value, a copy of the resource could migrate with the component. This would leave a copy of the resource at the source location such that other components referencing the resource wouldn't have to change their bindings. Of course, if the resource isn't transferable, the migrating component would have to either keep the binding across the network or simply remove the binding.

If a component has a reference to a resource that is bound by identity, the resource should either migrate with the component, or the resource should stay and be referenced across the network. If the resource is transferable and migrates with the component, the question is what happens to other components' bindings to the resource. These bindings can either be removed or they can be maintained across the network.

Using the introduced terms to describe the object movement in RMI, referenced objects are bound by value and moved by copy. However, server objects are bound by identity and referenced across the network. The same holds for aglets, where referenced aglets are bound by identity and referenced across the network, whereas other objects are bound by value and moved by copy. Telescript agents use binding by identity to referenced objects and resources. Upon migration some of the objects will migrate with the agent and all other bindings will be removed. Bindings from other agents or places to the migrated objects will also be removed. The class definitions used in a system can be seen as resources that are bound by type.

The different combinations of bindings and resources and the way they are handled at migration, are summarised in table 3.1.

3.4.3 Requirements Concerning Binding Handling

The different ways of handling bindings to resources and components at migration give rise to the following requirements:

- It should be possible to model by-type bindings.
- It should be possible to model that a referenced resource or component migrates together with a component.
- It should be possible to model that a copy of a referenced resource or component migrates together with a component.
- It should be possible to model that a component keeps a reference across the network to a resource or a component, which doesn't migrate together with the component.
- It should be possible to model that a reference to a resource or a component, which doesn't migrate together with the component, becomes invalid when the component moves.

3.5 Security

In all technologies where code moves between hosts, some kind of security mechanism must be present to protect the host against malicious code. If a component carries confidential data, there should further be some way of protecting the component from unauthorized modification or analysis of a malicious host. The described technologies all have some security mechanisms to protect a host, but none to protect the moving components.

In the Telescript technology an agent needs a permit to move to a new place or a new region. This permit will be given depending on the agent's authority. An agent may be denied access. An agent can also be allowed access to a place but be given limited capabilities, e.g. it won't be allowed to create new executing components or modify certain resources.

The type of the binding and the type of the referenced resource:	**The way the binding is handled when a component moves:**	**Examples of bound resources and components treated in this way:**
Binding by type	The referenced component stays. The moving object will be rebound to a similar resource at the destination.	- Java classes - Resources available at every location
Binding by value, transferable resource	The referenced component stays but a copy is being moved. The binding in the moving component is replaced with a binding to the copy.	- Java objects
Binding by value, non-transferable resource or Binding by identifier	The referenced component stays. The moving component will keep the binding across the network.	- RMI server objects - Aglets
Binding by value, non-transferable resource or Binding by identifier	The referenced component stays. The binding in the moving component will be deleted.	- Telescript objects that aren't owned by the moving object - Resources that only are accessible from the old location
Binding by identifier, transferable resource	The referenced component moves together with the moving object.	- Telescript objects owned by the moving agent

Table 3.1: When a component moves, its bindings to other components or resources can be treated differently. The table lists the different ways of handling the bindings and the cases where it is used.

Java also has a security mechanism that can protect a JVM against malicious code, again by limiting the capabilities of unauthorized code. Code fetched from the internet will generally not have access to all resources, opposed to code located in the local file system.

The security concerns give the following requirements to a modelling language:

- It should be possible to deny an executing component access to a location.
- It should be possible to limit a components capabilities regarding access to resources and creation of new executing components.

3.6 Openness

The Jini technology is designed for building open systems where components dynamically can be connected to the system and where components can become aware of each other. This is the kind of mobility where communication links are being moved, created or destroyed.

All components in a Jini system will from the beginning know how to contact the lookup service. A component may not know any other services from the beginning but it can become aware of other services by looking them up in the lookup service. The component can receive references to other services (the proxy objects) and will thus become able to contact and interact with the services. I.e. contact information is communicated between the components, and new communication links are being created in this way.

In UPnP the system will take care of generating a new IP address when a component is added to the system, i.e. a new location name is generated. When a new component is generated in Java and in the agent technologies, it will be given a new unique name, which is used by other components that want to interact with the new component. So new names, or information needed to communicate with a component, can be created dynamically.

This gives the following requirements to a modelling language:

- The language should allow dynamic creation of new names.
- The language should allow names (or what else is necessary for establishing new communication links) to be communicated between components.

3.7 Communication

The components in a system need to interact/communicate, and this interaction takes many forms in the described technologies.

3.7.1 Communication Forms in the Technologies

The objects in Java interact mainly by invoking each other's operations. If the objects are in different locations, an RMI call will be used to invoke the operation. An object needs a reference to another object in order to call its operations, but then it doesn't matter whether the object is local or remote. However, to obtain a reference to a remote object, it can be necessary to know the location of the object, because this location is needed before the registry can be contacted.

Objects in the same location also have the possibility of interacting through shared variables, whereas components in different locations can use network sockets to exchange messages/data. When socket communication is used, an object in one location will send a message, and an object in another location will be waiting to receive the message. The sender of a message needs to specify both the destination location and the port number where the receiving process is waiting for incoming messages.

Jini uses the same forms of interaction as Java, but a component doesn't need to know the location of a service in order to obtain a reference to it; it only needs to know the interface of the service. Components can also interact by means of distributed events. This is the multicasting of a message from one component to all the components that have registered interest in a certain event. Jini also uses multicast as part of the discovery protocol.

Aglets can communicate using synchronous or asynchronous message passing. An aglet can also request that another aglet takes certain actions (e.g. migration) by invoking an appropriate operation. Every aglet has a unique identifier which is used to obtain a proxy object through which all communication goes.

Telescript agents in the same place can invoke each other's operations as well as the place's operations. Agents in different locations can interact by message passing. Agents get a reference to each other when they meet and this reference is used when invoking operations. An agent can name whom to meet with by giving its telename or class name, that is, it can ask for a meeting with an agent of a given name, a given authority or a given class.

3.7.2 Classification of the Communication Forms

Most of the forms of component interaction in the above is based on *one-to-one* communication. A component interacts with one other component by invoking one of its operations or by sending a message to it. The other component is uniquely identified through a network address or through a reference, which holds the unique name of the component.

However, some of the technologies allow a component to specify interaction with just anyone out of a group of components; this is *one-to-one-of-many* communication. This communication form is used in the Telescript technology, where an agent can decide to meet and interact with anyone of a certain authority or class name. Likewise in Jini, where a component can decide to interact with any service with a given interface.

Jini also uses *one-to-many* communication where a component sends the same message to several other components. This is the case with the multicasting of event messages and the multicasting in the discovery protocol. In the first case the receiving components are uniquely identified components, whereas in the latter case the receiving components will be all the components in a certain part of the network.

Besides of this classification of communication forms into one-to-one, one-to-many and one-to-one-of-many, the communication forms can also be split into *one-way* and *two-way* communication. An operation call will result in a request being transmitted in one direction and a result (possibly a null result) being transmitted in the other direction. This is two-way communication. The message passing used between aglets and between sockets in Java, is one-way communication.

A further division into *synchronous* and *asynchronous* communication forms can be made. Operation calls are synchronous in all the described technologies, i.e. the component calling the operation will stop executing until a result has been returned. However other technologies allow asynchronous operation calls, e.g. CORBA [Emm00]. Using asynchronous operation calls, the calling component will continue execution as soon as the request has been issued. This allows concurrent processing of multiple requests.

For one-way communication both synchronous and asynchronous forms exist in the described technologies, e.g. aglets. In the synchronous form, the sender of a message can't continue execution until the message has been received. In the asynchronous form the sender continues execution as soon as the message has been sent.

Using the terms introduced above, the invocation of an operation can be described as one-to-one, two-way, synchronous communication. The interaction with a Telescript agent or Jini component that fulfils some requirement can be seen as one-to-one-of-many, two-way, synchronous communication. Distributed events is a form of one-to-many, one-way, asynchronous communication. Interaction through shared memory can be seen as one-to-one-of-many, one-way, asynchronous communication, and message passing is one-to-one, one-way, synchronous or asynchronous communication.

There is one more aspect of the communication that needs to be addressed: The location dependency of the interaction. A component may need to know the location of another component in order to interact with it, but this will depend on the kind of interaction. For some forms of interaction it will even be necessary for the component to be co-located with the other component before they can interact, as discussed in section 3.2.

When it comes to the addressing of the component to interact with, RMI provides location independent addressing. If a component has a reference to another component, it can invoke its operations without knowing the location of the component. It is the same in the aglet technology, where an aglet can interact with another aglet through a reference to a proxy object, no matter the aglet's location. In all other cases the addressing is location dependent. A Java object needs to know the location of a registry before they can interact, and message passing also takes that the sender knows the location of the receiver.

3.7.3 Requirements Concerning Communication

The different communication forms used in the described technologies give rise to the following requirements to a modelling language:

- The language should provide a way of addressing whom to communicate with, allowing both one-to-one, one-to-many, and one-to-one-of-many communication.
- The language should support both location dependent as well as location independent addressing.
- It should be possible to model both one-way and two-way communication.
- It should be possible to model both synchronous and asynchronous communication.

3.8 Synchronization

When several executing components are present in a system, they may need to synchronize their actions. Some sort of synchronization is already involved when two components communicate: A client requesting an operation has to wait until a reply is returned, and the receiver of a message has to wait until a message has been sent. This synchronization happens implicitly, but other cases may need a more explicit synchronization.

Generally, there are two basic kinds of synchronization [And00] between executing components: *mutual exclusion* and *condition synchronization*. Mutual exclusion is the problem of ensuring that certain operations do not execute at the same time. E.g. operations that operate on the same data or uses the same resources may need to execute under mutual exclusion. Condition synchronization is the problem of delaying the execution of a component until a given condition is fulfilled.

This gives the following general requirements to a modelling language:

- It should be possible to model mutual exclusion.
- It should be possible to model condition synchronization.

3.9 Failure Handling

In a distributed system with several executing components, it may happen that one component fails without all of the system failing. This is known as partial failure. If a single component fails or becomes overloaded or the network breaks down, other components should still be able to proceed despite the failure. The described technologies provide different mechanisms for handling partial failure.

The Java based technologies use exceptions to allow a component to become aware of failures. A client requests an operation and is prepared for the failure of the operation. In case of failure, an exception will notify the client of the details of the failure, and the client can then act upon it. E.g. when a client makes an RMI request, it should be prepared for a `RemoteException` indicating that the server object has failed or couldn't be contacted.

Jini operates with lease based service registration and access. This allows the lookup service to become aware that a component has become unavailable (in the case where a registration isn't renewed when the lease period expires) and the registration can thus be removed to prevent further errors in that regard.

When a component has become aware of an error, it can start acting upon it and try to recover. Transactions are part of this failure handling. If one operation fails, the other operations in the transaction will become aware of this, and will undo their work. This happens under mutual exclusion with other operations trying to access the data being operated on. So transactions are just a mechanism for letting other operations/components become aware of a failure and act upon it (i.e. undo the operation).

To allow the modelling of a system where partial failure can arise and can be handled by the other components, the following requirements must be posed on a modelling language:

- It should be possible to model failure in a component.
- The language should provide a way of letting other components become aware of failure in a component.

3.10 Object Properties

All the described technologies are object-oriented and this gives some extra aspects to consider. Object oriented programming introduces the concepts of encapsulation, interfaces, object instantiation, inheritance, and polymorphism [Wan99].

An object provides *encapsulation* by grouping together data and operations on these data. In Java this takes the form of fields and methods operating on the fields. The data and the operations are the *members* of an object.

Only some of the members of an object are accessible from other objects. These publicly accessible members constitute the interface of the object. In Java this takes the form of fields and methods being declared either *private* or *public*. Private members can be accessed only from within the object, whereas public members are accessible by every object. Java also allows the intermediary modifiers "protected" and "package", which make the members accessible only by certain objects.

The members of an object are defined in a class, which can be *instantiated* to create one or more objects with the given definitions. An object will at creation be given initial values of its data members. This relation between an object and a class has already been treated in section 3.1, which describes that a code component can be used to create several passive components.

Inheritance allows a class to reuse definitions from another class; a subclass inherits members defined in its superclass and can add members of its own. In Java this is done through class extension, where a class can extend another class. Java only allows *single inheritance*, i.e. inheritance from a single class, but generally a class can inherit members from multiple classes.

Objects created from different classes can still have operations with the same signature. This allows the definition of *polymorphic* programs or operations that can operate on different kinds of objects. Java supports polymorphism through interface definitions and dynamic method overriding. An interface prescribes that a class implementing the interface has certain operations. Method overriding allows a subclass to define an operation with the same signature as an operation in a superclass but with a different implementation.

Object-orientation gives rise to the following requirements to a modelling language:

- It should be possible to model encapsulation of data and operations working on the data.
- It should be possible to model that some data or operations are private to a component, i.e. can't be accessed from other components.
- A component should be able to inherit code from other components.
- The language should allow polymorphism.

Chapter 4

Formal Languages with Mobility

There exist several formalisms that allow one to model and reason about systems with mobile components. One approach is to model mobility as the changing of communication paths; another approach is to use explicit notions of location and migration of processes between these locations.

This chapter will present four formal languages that can express mobility: the π-calculus, the join-calculus, formalizations of the actor model, and the ambient calculus. The basic π-calculus and the actor model express mobility as the changing of communication paths, whereas the join-calculus, the ambient calculus, and an extension of the π-calculus, Nomadic π, have explicit location concepts.

All the formal languages build on a minimal set of primitives; they focus on the communication between processes and abstract away other computation issues. A formal semantics and equivalence relations have been defined for each language, allowing one to reason about systems. The equivalence relations are defined in terms of the observable behaviour of a system, which in most cases means the possible communication of the system with the environment. This provides a formal framework for modelling, reasoning about and verification of systems and programs.

A full formal description of the languages will not be given; only their syntax and the interesting part of their semantics will be presented. This is to focus on how the different formalisms work and what they can do in comparison to each other. For a full formal description of the languages is referred to the literature.

4.1 π-Calculi

The π-calculus [San01][Sew01] is a process calculus that allows dynamic creation of communication paths.

Like in all other process calculi, the basic entities are processes, which run concurrently and communicate through channels. The channels connect the processes in a system and they thus define the network topology of the system. The π-calculus allows dynamic creation of channel names and transmission of these names, thereby enabling processes to establish new connections. This dynamic changing of a process's communication links can be used to model the movement of a process; when a process moves, its neighbourhood will change and thus its connections to other processes.

There are several versions of the π-calculus. *Monadic* π-calculus is the basic form, which allows only a single channel name to be sent in a message. *Polyadic* π-calculus extends the monadic π-calculus to allow a tuple of channel names to be sent in a single message. The *higher-order* π-calculus also allows processes to be sent in a message, and it can thus model mobility more directly. The expressiveness is however the same.

Nomadic π [SWP99] is an extension of the π-calculus with a more explicit notion of mobility. It adds sites and agents that can migrate between sites, and it operates with two levels of abstraction in its communication mechanisms. Otherwise it has the same basic properties as the π-calculus.

4.1.1 Monadic π-Calculus

A Process in the monadic π-calculus is composed from other processes and from communication actions on named channels. A communication action can be an input or an output of a single value to a channel. Channel names are the only values in the π-calculus, allowing one to focus on the communication.

The communication can be either synchronous or asynchronous; the π-calculus exists in both versions. An asynchronous version will be presented here because this has become normal and it is closer to the communication mechanism in real distributed systems.

A basic monadic asynchronous process calculus is given by the following abstract syntax:

$$P ::= 0 \ \mid \ P|P \ \mid \ c\langle v\rangle \ \mid \ c(x).P \ \mid \ (\nu c)P$$

0 is the inactive process, $P|P$ is the parallel composition of processes and $(\nu c)P$ introduces a new channel c with the scope restricted to P. A channel name can however be communicated out of its scope. $c\langle v\rangle$ is the asynchronous output of v on channel c, and $c(x).P$ is the input of a value on channel c, after which the process will behave as P with the received value substituted for x. A synchronous process calculus would not only allow process continuations after an input action but also after an output action.

The calculus can be extended with a way of expressing iteration or recursion, which allows one to model infinite computations. Most process algebras use replication $!P$ which is equivalent to $P|!P$. Recursion is also possible with the use of process variables x and a recursion operator $rec\ x.P$ where the process P is defined in terms of x.

Many versions of process calculi exist with slight variations. The notation used for writing input and output operations vary. Some versions also have a form of choice operator ($P+Q$) which allows non-determinism. However, an explicit choice operator isn't always necessary and internal choice can often be expressed in other ways.

An operational semantics for the π-calculus can be given as a structural equivalence relation ($\equiv$) and a system of labelled transition relations ($\xrightarrow{\mu}$) between processes. The structural equivalence relation should specify that parallel composition is commutative and associative, that restriction is commutative and that a restriction can be extended to cover processes where the restricted name isn't used ($P|(\nu x)Q \ \equiv (\nu x)(P|Q)$ if $x \notin \text{fn}(P)$). The labels in the transition system specify the communication with the environment. The possible labels μ are:

τ	internal communication
$c(v)$	input of v on channel c
$c\langle v\rangle L$	output of v on channel c. If the name v is sent out of its scope, it is contained in the set L, i.e. L is a singleton. Otherwise L is empty.

The set of labelled transition relations is specified by a set of axioms and rules. The following illustrates the idea:

$$
\begin{array}{ll}
\mathit{Out} & c\langle v\rangle \xrightarrow{c\langle v\rangle\emptyset} 0 \\
\mathit{In} & c(x).P \xrightarrow{c(v)} P\{v/x\} \\
\mathit{Open} & \dfrac{P \xrightarrow{c\langle x\rangle\emptyset} P'}{(\nu x)P \xrightarrow{c\langle x\rangle\{x\}} P'} \\
\mathit{Com} & \dfrac{P \xrightarrow{c(x)} P' \quad Q \xrightarrow{c\langle x\rangle\{x\}} Q'}{P|Q \xrightarrow{\tau} (\nu x)(P'|Q')}
\end{array}
$$

The rule *Open* specifies scope extrusion, i.e. the communication of a name out of its scope. The rule *Com* specifies how an input and an output action synchronizes to give an internal action. A name sent out of its scope will be restricted to the two communicating processes. Another *Com* rule should specify the case without scope extrusion. Further more there should be rules for specifying parallel composition, communication of non-restricted names out of restrictions, and a rule for structural equivalence of processes.

A large mathematical framework has been developed for the π-calculus to allow one to reason about processes. An important concept is that of observational equivalence where two processes are considered equivalent if their possible behaviours, as observed from the environment, are the same. The behaviour that can be observed from the environment is the sending and receiving of messages on unrestricted channels. These messages can be picked up or sent from the environment. Two processes are considered equivalent if they, placed in an arbitrary environment, allow the same sequences of messages to be exchanged with the environment.

4.1.2 Polyadic and Higher-Order π-Calculi

Polyadic π-calculus allows tuples of names to be communicated and adds the concept of *abstractions* and *concretions*. Abstractions are processes with formal parameters. An abstraction can be given actual parameters to give a concrete process. An abstraction takes the form of a *constant* D, which is defined as a process with some formal parameters $\tilde{x}$:

$$D \stackrel{\text{def}}{=} (\tilde{x})P$$

The process P would refer to the names in $\tilde{x}$. A concrete process with the actual parameters $\tilde{y}$ is created with $D\langle\tilde{y}\rangle$. The transition rule for the use of constants is:

$$\frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\mu} P'}{D\langle\tilde{y}\rangle \xrightarrow{\mu} P'} \quad \text{if } D \stackrel{\text{def}}{=} (\tilde{x})P$$

The other transition rules for the monadic π-calculus can easily be generalised to the polyadic π-calculus by replacing names with tuples of names. Also, the set L in the label $c\langle v\rangle L$ can now hold more than one name because several names can be transmitted out of their scope in a single output action.

In the higher-order π-calculus, processes can be transmitted on channels. The processes may be parameterized, as the abstractions in the polyadic π-calculus. When a process has been transmitted, it can begin its execution. The following example illustrates how a process P can be sent from one process to another and then be executed:

$$c\langle P\rangle \mid c(X).X \xrightarrow{\tau} P$$

The transition rules for the higher order π-calculus is basically the same as for the polyadic π-calculus.

4.1.3 Nomadic π

The Nomadic π calculus is an extension of the asynchronous polyadic π-calculus with notions of sites, agents and migration. It is intended as the basis for a programming language for distributed systems. The calculus is described in [SWP99] and is implemented in the Nomadic Pict System [Woj00].

A system consists of named *sites* (locations) and *agents*, which can migrate between the sites. An agent has a unique name and a body consisting of a process. Agents can be created dynamically:

`agent` $a = P$ `in` Q

This will start a new agent on the current site, with the body P running concurrently with the process Q. An agent can migrate to another site by executing:

`migrate to` $s \rightarrow P$

in its body. This will move all the processes in the agent's body to site s, whereafter process P will be executed together with the other processes. As is the case with channel names, agent names may be transmitted out of their scope.

The introduction of locations adds the aspect of location dependency or independency in the communication forms. Nomadic π has both a location dependent as well as a location independent addressing form, but the two forms are intended for use at different abstraction levels. The location independent addressing is intended for use at the programming level whereas the location dependent addressing is needed at the runtime system level. A translation should be given of the location independent communication into constructs that only use location dependent communication. The translation allows one to specify an appropriate infrastructure that supports the location independent communication.

Some of the different communication constructs are:

$c?x \ = \ P$	input from channel c
$c!v$	output to channel c in the same agent
$\langle a \rangle c!v$	output to channel c in agent a on the same site
$\langle a@s \rangle c!v$	output to channel c in agent a on site s
$c@a!v$	location independent output to channel c in agent a.

The location dependent constructs will fail silently if the channel or the agent isn't where it is expected to be. New channels are created with the construct `new` *chan* `in` P and replicated communication is written with a "$*$" in front of the communication construct.

4.2 Distributed Join-Calculus

The join-calculus[FG96] is a process calculus quite similar to the π-calculus. It builds on asynchronous communication on channels and it has names as the only values. The expressiveness is the same as for the π-calculus and mobility is modelled in a similar way with the exchange of channel names. However, the creation of a new channel name, the receipt of a message on that channel, and replication of the process that performs this input action, is combined to a single receptor definition. Such a receptor definition is similar to the definition of an operation and this gives the calculus a functional programming style.

The distributed join-calculus[FGL$^+$96] is an extension of the join-calculus with notions of location, migration and failure. The distributed join-calculus is the basis for the join-calculus language [FM97], which is a simple but high-level programming language for distributed systems. It uses lexical scoping and location independent addressing, which is also reflected in the calculus.

The semantics of the join-calculus and the distributed join-calculus is given in terms of a chemical abstract machine (CHAM). The chemical abstract machine [BB92] is a model for concurrent computations and it will be shortly introduced below. The join-calculus is induced by the reflexive CHAM, which is a particular CHAM that specifies how particular computations are carried out. The distributed join-calculus is induced by the distributed reflexive CHAM (DRCHAM). These two CHAMS will also be presented below together with the syntax for the two calculi.

4.2.1 The Chemical Abstract Machine (CHAM)

The chemical abstract machine is a general model for how concurrent computations can be carried out. The CHAM is described with terms from the chemistry because the mechanisms of the CHAM resemble the mechanisms in a chemical solution. According to the model, the concurrent components of a system float around between each other and have the possibility of interacting when they come in contact. The interactions can happen concurrently as long as they don't involve the same components.

The components of a system are called *molecules* and they float around in a *chemical solution*. The molecules can interact with each other according to some given *reaction rules*, which specify how the molecule structure changes when molecules react. Not all molecules can interact with others, but *heating rules* can specify how a molecule can be heated into several simpler submolecules, and *cooling rules* can specify how a set of molecules can be cooled down to a single molecule. A solution that can't be cooled any more (i.e. no cooling rule applies) is called *frozen*.

A chemical solution is represented by a multiset of molecules, $S = \{\!|m_1, m_2, \dots|\!\}$. The molecules are terms of some algebra, but the exact syntax of the molecules will vary from CHAM to CHAM. A solution can be wrapped in a *membrane* $\{\!|.|\!\}$ and can itself be considered a molecule. A solution can thus appear as a subsolution of another molecule.

A CHAM has associated rules, which specify how the molecules of the specific CHAM can interact and be transformed. The rules have the general form:

$$m_1, m_2, \dots, m_k \to m'_1, m'_2, \dots, m'_l$$

Only the molecules that are affected by the rule are mentioned. A set of general laws define the overall behaviour of a CHAM, i.e. how the transformations specified by the rules can take place.

The general laws basically state that rules can be applied to molecules inside any solution, irrespective of the other molecules present in the solution and irrespective of the context of the solution. This will result in the molecules at the left-hand-side of the rule being replaced with the molecules at the right-hand-side of the rule.

The rules of a specific CHAM are categorised into heating rules($\rightharpoonup$), cooling rules($\rightharpoondown$) and reaction rules($\to$) as mentioned above. Heating or cooling of molecules is often reversible as illustrated by the following example of a heating/cooling rule pair. The rule expresses that a molecule consisting of a parallel composition can be heated into its subcomponents that will become new molecules in the solution, ready to interact with other molecules:

$$p|q \rightleftharpoons p, q$$

Since multisets are unordered, the parallel composition is naturally associative and commutative when the semantics is specified in this way.

Reversible heating/cooling rule pairs represent the syntactical rearrangements of terms in the solution. If a molecule can be transformed into another using only heating and cooling rules, it corresponds to the two molecules being structurally equivalent.

The above is the general description of how a CHAM works. A concrete CHAM will specify the syntax of the molecules and give a set of rules. A CHAM can be used to describe the semantics of a concurrent language. The language constructs make up the molecules of the cham. The reaction rules and the general laws of the CHAM define the semantics of the language.

4.2.2 The Reflexive CHAM

The reflexive CHAM[FG96] gives a specific syntax for the molecules and adds a few rules to the generic CHAM. The rules allow a reaction to extend the CHAM with new kinds of molecules and definitions of *reductions* of these new kinds of molecules. The reduction definitions are used together with a meta-reaction rule to specify how the new kinds of molecules can be transformed into other molecules.

The molecules in the reflexive CHAM are:

Pending messages, $x\langle\tilde{v}\rangle$, meaning that the names $\tilde{v}$ are sent on the channel with the name x. Channels will be referred to as *ports* because they constitute the interface (access points) of a particular process. The port name of a pending message determines the kind of molecule and the possible kinds of reductions of the molecule. A port has an arity telling the number of names in a message on that port, and it will be assumed that this arity is respected in all messages.

Defining molecules, `def` D `in` P, where D defines one or more new reductions of new kinds of molecules, i.e. of pending messages on ports of new names. The new port names may be used in the molecule P. A reduction definition is of the form $J \triangleright P$ as will be described below. New reduction definitions can't be added for existing port names; a port name is defined together with the reduction working on messages on that port.

Compound molecules, $m_1|m_2$, where the sub-molecules m_1 and m_2 are combined with the join-operator. This is the concurrent composition of components.

To allow the CHAM to be extended with new definitions of reductions, a reflexive CHAM is represented by a higher-order solution, $\mathcal{D} \vdash \mathcal{P}$. The multiset $\mathcal{D}$ contains the currently defined reductions and the multiset $\mathcal{P}$ defines the molecules (processes) in the solution. The rules of the reflexive CHAM operate on such higher-order solutions.

$\mathcal{D}$ contains reduction definitions of the form $J \triangleright P$. The join pattern J describes a compound molecule consisting of pending messages on specific port names. The transmitted values in a pending message are represented by formal parameters. An example join pattern could look like this: $port_1\langle x_1, x_2\rangle \,|\, port_2\langle y\rangle$. The reduction definition $J \triangleright P$ specifies the transformation of a compound molecule of the form J into a molecule of the form P. The actual reaction of molecules, as specified by a reduction definition, is carried out by the following meta-reaction rule for the reflexive CHAM:

$$J \triangleright P \vdash J\sigma_{rv} \rightarrow J \triangleright P \vdash P\sigma_{rv}$$

The substitution σ_{rv} instantiates the formal parameters of J to the actual transmitted values.

The following heating/cooling rule pair expresses how defining molecules can introduce reduction definitions into the reflexive CHAM:

$$\vdash \texttt{def}\ D\ \texttt{in}\ P \rightleftharpoons D\sigma_{dv} \vdash P\sigma_{dv}$$

The substitution σ_{dv} instantiates the port names defined in D to distinct, fresh names. D can be a conjugation of several reduction definitions, and another pair of heating/cooling rules specify how such a conjugation can be decomposed into definitions of a single reduction. A last heating/cooling rule pair specifies how compound molecules can be heated into sub-molecules, which reversely can be cooled down to compound molecules.

The molecules of the reflexive CHAM constitute the terms of the join-calculus. The semantics is given by the rules of the reflexive CHAM. The terms of the join-calculus are processes specified by the following abstract syntax:

$$\begin{aligned} P &::= x\langle\tilde{v}\rangle \mid \texttt{def}\ D\ \texttt{in}\ P \mid P|P \\ D &::= J \triangleright P \mid D \wedge D \\ J &::= x\langle\tilde{v}\rangle \mid J|J \end{aligned}$$

The process P in the definition $J \triangleright P$ is called the guarded process of the join-message J.

The join-calculus doesn't have a separate input action, a restriction operator or a replication operator as the π-calculus. These constructs are instead combined in the definition $\texttt{def}\ J \triangleright P\ \texttt{in}\ P'$. This construct corresponds to a replicated input from the ports in J followed by the process P. The construct also introduces the new port names in J and restricts their scope to P and P'. Since there is no separate input action, the definition construct defines the only process that can input from the defined ports.

When a message is sent to the ports defined in J, the process P will be started, with the actual values from the message substituted for the formal parameters. This is very similar to the invocation of a function, and the reduction definition corresponds closely to a function definition in a functional language. This gives the join-calculus a functional style.

Syntactic sugar is added to the join-calculus to support sequencing of function calls and two-way communication where a function returns a value. To express the sequencing in the pure join-calculus, a continuation passing style is used, where a continuation channel κ is passed together with the other arguments to a port. A message will be returned on this channel upon completion, and this will start the guarded process P of that port:

$\texttt{def } f\langle x,\kappa\rangle \triangleright \ldots \kappa\langle\rangle \ldots \texttt{in}$
$\quad \texttt{def } \kappa\langle\rangle \triangleright P \texttt{ in } f\langle\tilde{v},\kappa\rangle$

Syntactic sugar allows this to be written as:

$\texttt{def f}(x) \triangleright \ldots \texttt{reply to f} \ldots \texttt{in f}(\tilde{v}); P$

In the case of two-way communication, where f returns a result $\tilde{V}$ to port κ for use in process P, a let-binding is introduced to express this:

$\texttt{def f}(x) \triangleright \ldots \texttt{reply } \tilde{V} \texttt{ to f} \ldots \texttt{in let } \tilde{u} = \texttt{f}(\tilde{v}) \texttt{ in } P$

This makes the functional style even more obvious.

4.2.3 The Distributed Reflexive CHAM

The distributed join-calculus [FGL+96] is an extension of the join-calculus with a notion of location and migration. The calculus is induced by the distributed reflexive CHAM (DRCHAM) which builds on the reflexive CHAM.

A DRCHAM is a multiset of reflexive CHAMS and as such it consists of several local solutions, each having a unique location name and its own set of reductions. The local solutions can evolve separately according to the rules of the reflexive CHAM, but they can also communicate and use each other's reductions. A solution can also migrate to another solution and become a sub-solution of that one.

A local solution represents a location with its own definitions and running processes. The locations can be nested and are organised in a tree. A location is represented by a labelled solution $\mathcal{D} \vdash_\phi \mathcal{P}$, where the label ϕ is a string of location names. ϕ reflects the location's position in the tree and it holds the names of the locations met on a traversal from the root of the tree to the current location. The labelled solutions in a DRCHAM are separated with a '$\|$'.

The syntax of the molecules is extended to allow creation of new locations. A location with the name a and the local definitions D and processes P is created with the following definition:

$$a[D : P]$$

The new location will become a sublocation of the location where it is defined. This is expressed in the following pair of heating/cooling rules for the DRCHAM:

$$a[D : P] \vdash_\phi \quad \rightleftharpoons \quad \vdash_\phi \ \| \ \{\!|D|\!\} \vdash_{\phi a} \{\!|P|\!\} \qquad (a \text{ frozen})$$

The side condition means that if there are any sublocations of a, they have all been cooled down to location definitions in D.

Locations can communicate by using each other's defined reductions. A pending message on a port can be sent between locations to the location where the port name is defined. This is expressed in the following reaction rule:

$$\vdash_\phi x\langle\tilde{v}\rangle \ \| \ J \triangleright P \vdash_\psi \quad \rightarrow \quad \vdash_\phi \ \| \ J \triangleright P \vdash_\psi x\langle\tilde{v}\rangle \qquad (x \text{ is defined in } J)$$

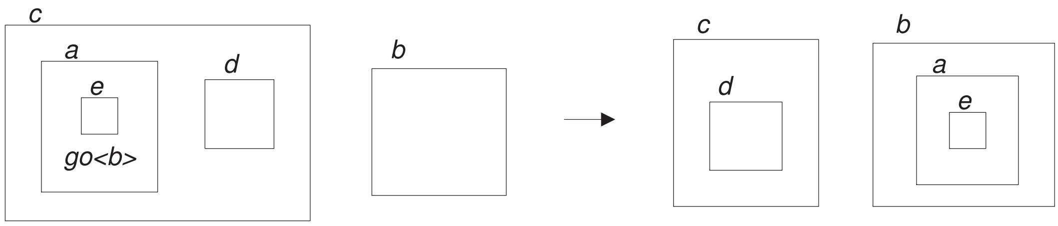

Figure 4.1: Location a migrates to location b, bringing all its sublocations. The destination location can be any location in the location hierarchy, except for a sublocation of a.

Every port name should be defined in at most one location to make this message transport deterministic.

Finally, a location is allowed to migrate to another location to become a sublocation of that location. When the primitive $\mathtt{go}\langle b, \kappa\rangle$ is executed in a location, the entire location together with all its sublocations will migrate to location b (see figure 4.1), as expressed by the following rule:

$$a[D : P|\mathtt{go}\langle b, \kappa\rangle] \vdash_{\phi} \ \| \ \vdash_{\psi b} \quad \rightarrow \quad \vdash_{\phi} \ \| \ a[D : P|\kappa\langle\rangle] \vdash_{\psi b}$$

When a has arrived at the new destination, the continuation κ can start other computations. Location names are treated as first-class values to allow location names to be sent in messages and become known outside their scope.

The distributed join-calculus also has primitives that allow the modelling of failure of a location. A marker sign Ω is added in front of a location name to indicate that the location has failed. Computations can't continue in a location that has failed, and all the reaction rules are thus given a side condition such that messages and sublocations can't leave a failed location and reductions can't be performed in the location. This will also apply to all the sublocations of a failed location.

The execution of the primitive $\mathtt{halt}\langle\rangle$ will cause a location to fail, while the primitive $\mathtt{fail}\langle a, \kappa\rangle$ can be used for error recovery. The primitive can detect that location a has failed in which case it will trigger the continuation κ.

As in the π-calculus, an observation equivalence can be defined between processes in the join-calculus. A process will be characterised by its ability to emit messages on certain port names since this emission of a message can be observed by the environment.

The definition of observation equivalence is based on the definition of a reduction relation ($\longrightarrow$) on processes. A process P can be reduced to P', written $P \longrightarrow P'$, if the solution containing only P can be transformed into a solution containing only P' by a sequence of heating and cooling steps and a single reaction:

$$P \longrightarrow P' \ \stackrel{\text{def}}{=} \ \emptyset \vdash \{\!|P|\!\} \ (\rightleftharpoons^{*}\rightarrow\rightleftharpoons^{*}) \ \emptyset \vdash \{\!|P'|\!\}$$

Two processes are considered observationally equivalent if they both can be transformed into a process with a pending message on a certain port, by a sequence of reduction steps.

4.3 Actors

The actor model is a general model for structuring concurrent systems. The model describes a system as being built of autonomous interacting components, named actors. The model is good at describing open systems, where components may be created, replaced or connected with other components.

The actor model has been the basis for several programming languages, e.g. SALSA and THAL [Kim97] [Rad99] [VA01]. The model has also been formalized and given a formal semantics that allows reasoning about and verification of actor systems.

In the following I will describe the actor model and present two different approaches to formalizing the model. The first approach, [ATZ01] [Agh99] [AMST97], is based on the extension of a functional language with some simple actor primitives. The second approach, [GZ01], gives a syntax and a semantics that are more similar to that of process calculi. Finally I will look into some of the extensions to the actor model that are made in some actor languages.

4.3.1 The Actor Model

The actor model is a model of coordination between autonomous interacting components. The components are named actors and they run concurrently and interact by means of message passing. An actor has a state, a set of operations on the state, and a thread of control. The thread is not active all the time but is triggered by the receipt of a message. See figure 4.2 to see the components of an actor.

Actors communicate by asynchronous one-to-one message passing. Each actor has a unique name (address) that is used to specify the recipient of a message. Each actor also has a mailbox where messages are buffered until the actor is ready to receive the messages, one at a time. Like in any other model based on asynchronous messages, no ordering of the messages can be assumed. I.e. if an actor sends two messages, one after the other, the messages can arrive in any order to the receiver. However, in the actor model the message passing is fair in the sense that all messages will eventually be received.

An actor's behaviour is a function of the incoming communication. The receipt of a message will trigger some action in the actor. In most actor languages, the receipt of a particular kind of message is analogue to the invocation of a particular operation. The names of the message kinds/operations are not unique but can be reused in several actors. To invoke an operation in an actor, the sender thus needs

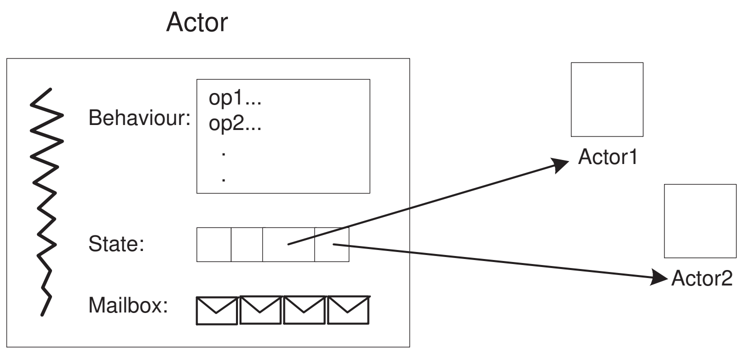

Figure 4.2: An actor has a behaviour, a state, a mailbox from where it receives messages, and a thread of control. The behaviour specifies how the actor should react when it receives messages of certain kinds.

to know the unique actor name plus a general operation name. This is opposed to the process calculi where an operation is modelled by a unique channel name that will be different for each component that offers the operation. In the actor model, actor names are first class values that can be sent in messages. Operation names are treated likewise in most actor languages.

A distinction between an *actor* and its *behaviour* is made. A behaviour defines the actions to be taken in response to a message. Such behaviours can be defined and actor instances can be created from these behaviours.

When an actor receives a message, it can in response

- *send* new messages to other known actors or itself
- *create* new actors with a specified behaviour
- *become* a new actor that is ready to receive the next message.

These three kinds of actions constitute a set of actor primitives that can be added to any sequential language.

When an actor has received a message, it will execute some of the above actions as defined by its behaviour. The actor will not be able to receive another message until it has executed a *become* action. This action changes the state and behaviour of the actor and makes it ready to receive a new message. That is, the actor is locked until a replacement behaviour is established that can take over the processing of the next message.

Some actor languages allow a *become* to be executed before the actor has finished executing all the other actions in response to the message. This frees the actor to receive a new message while the remaining actions are being executed. This allows multiple threads to be active inside an actor. Likewise, some actor languages [Rad99] allow all actions/statements to be executed concurrently but this demands that some form of intra-actor synchronization primitives are available. However, most actor languages allow only one thread of control in each actor.

Like in the join-calculus, the receipt of messages is implicit, i.e. there is no statement in the behaviour definition that expresses that a message should be received. Instead, the receipt of messages happens implicitly when an actor has executed a *become* and there are undelivered messages in its mailbox.

4.3.2 A Functional Actor Formalism

In [AMST97] and [ATZ01] a simple functional language is extended with actor primitives to give a simple actor language. An operational semantics is given in terms of a transition relation between actor configurations, and several forms of equivalences are defined.

Language Syntax

The actor language relies on a functional language to provide the basic computation facilities, whereas the actor primitives only are concerned with the coordination of concurrent components. The actor primitives are

$\texttt{send}(a, m)$	creates a new message with content m and receiver a.
$\texttt{letactor}\{x := b\}e$	creates a new actor with behaviour b. The name of the actor is assigned to x and can be used in the proceeding expression e.
$\texttt{become}(b)$	replaces the actor's behaviour with b and frees the actor to receive a new message.

This corresponds to the *send*, *create* and *become* actions of the actor model.

Actor behaviours, b, are defined as named λ-abstractions that can be applied to messages. When an actor receives a message, its current behaviour will be applied to the message. The behaviour can be a higher order function that can take several arguments, in the end giving a function that can be applied to messages. In this way, a state can be carried around in the first arguments.

The following example illustrates how an object with a state and operations on the state could be modelled. The example assumes that there are functions $\texttt{m}_1\texttt{?}$, $\texttt{m}_2\texttt{?}$, etc. for testing the kind of message, and constructors $\texttt{m}_1\texttt{()}$, $\texttt{m}_2\texttt{()}$, etc. for creating messages of certain kinds. The behaviour *Class* is defined as a recursive λ-abstraction that can be applied to a state s to give a function that accepts messages m. The actor *object* is created from this behaviour and a message is sent to invoke operation m_1:

$$
\begin{array}{l}
Class = \texttt{rec}(\lambda b.\lambda s.\lambda m. \\
\quad \texttt{if}(\texttt{m}_1\texttt{?}(m), \\
\quad\quad \ldots method_1 \ldots \texttt{become}(b(m_1(s, arg))), \\
\quad\quad \texttt{if}(\texttt{m}_2\texttt{?}(m), \\
\quad\quad\quad \ldots method_2 \ldots \texttt{become}(b(m_2(s, arg))), \\
\quad\quad\quad \ldots)))
\end{array}
$$

$$\texttt{letactor}\{object := Class(s_0)\}\texttt{send}(object, \texttt{m}_1\texttt{(}arg\texttt{)})$$

Semantics

The semantics of the language is specified in terms of a transition relation between actor configurations. An actor configuration is meant to represent a part of an open system, capable of exchanging messages with the surroundings.

An actor configuration consists of a collection α of named concurrently executing actors, a collection μ of messages in transit - to or from actors in the configuration, a set of receptionist names ρ, and a set of external actor names χ. A message with content v in transit to actor a is written $a\langle v\rangle$[1], and the entire actor configuration is written $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$.

The receptionists are those actors in the configuration that are known outside the configuration, and they thus define the interface to the environment. The externals are those actors in the environment that are known to the actors in the configuration. Both the set of receptionists and the set of externals may grow as the computation evolves.

The collection of actors, α, is represented as a map from actor names to actor states, written with the actor name as a subscript. The state of an actor tells whether the actor is busy processing a message or whether it is ready to receive a message. In the first case, the state is written $[e]$ where e is an expression representing the execution state. In the second case, the state is written (b) where b represents the actor's behaviour.

The transition relation $\mapsto$ specifies how an actor configuration changes when an actor executes an actor primitive or a construct from the functional language, or when an actor receives a message.

In order to define the transition relation, every actor expression is decomposed into a reduction context filled with a redex, written $R[redex]$. The redex identifies the next subexpression to be executed, and it will be either an actor primitive or a functional construct. The functional constructs will be reduced according to a reduction relation

[1]The literature uses the notation $< a \Leftarrow v >$ or $a \lhd v$ but I will use the notation $a\langle v\rangle$ in order to keep it similar to the other presented formalisms.

$\overset{\lambda}{\mapsto}$. This relation specifies the semantics of the sequential functional language and it is thus not interesting in this connection.

The transition relation between actor configurations is defined by a set of reduction rules. Some of the interesting rules are shown below:

`<rcv:a,v>`

$$\langle\!\langle\ \alpha, (b)_a \mid a\langle v\rangle, \mu\ \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle\ \alpha, [\mathtt{app}(b,v)]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

`<out:m>`

$$\langle\!\langle\ \alpha \mid \mu, m\ \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho'}_{\chi} \quad \text{if } m = a\langle v\rangle,\ a \in \chi,\ \text{and } \rho' = \rho \cup (\mathrm{FV}(v) \cap \mathrm{Dom}(\alpha))$$

`<in:m>`

$$\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle\ \alpha \mid \mu, m\ \rangle\!\rangle^{\rho}_{\chi \cup (\mathrm{FV}(v) - \mathrm{Dom}(\alpha))} \quad \text{if } m = a\langle v\rangle,\ a \in \rho,\ \text{and } \mathrm{FV}(v) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

`<fun:a>`

$$e \overset{\lambda}{\mapsto} e' \Rightarrow \langle\!\langle\ \alpha, [e]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle\ \alpha, [e']_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi}$$

`<bec:a,a'>`

$$\langle\!\langle\ \alpha, [R[\mathtt{become}(b)]\ ]_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \mapsto \langle\!\langle\ \alpha, [R[nil]\ ]_{a'}, (b)_a \mid \mu\ \rangle\!\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

The first rule says that when an actor receives a message, its behaviour is applied to the message content. The next two rules allow communication with the environment. Messages from the environment can enter the configuration if the messages are sent to one of the receptionists, and messages can leave the configuration if they are sent to one of the externals. The rule `<fun:a>` connects the transition relation with the reduction relation for functional constructs.

The rule for `become` expresses that a new anonymous actor is created to complete the execution of the current behaviour, whereas the original actor will move on to the receipt of a new message. This allows an actor to execute a `become` before it has finished all actions in response to a message, but there will still be only one active thread in an actor.

What is interesting about an actor configuration is its ability to communicate with the environment. Based on this idea, [AMST97] define an equivalence relation between actor configurations. The idea is that two configurations are equivalent if they have the same interface to the environment (receptionists and externals) and they allow the same sequences of message exchanges with the environment.

4.3.3 A Process Algebraic Actor Formalism

In [GZ01] an actor algebra using a "process algebra" style is defined and an operational semantics is given in terms of a labelled transition system and a structural equivalence relation between actor terms. The actor algebra describes only the communication and synchronization aspects of an actor system and does not deal with any sequential computation constructs, as was provided by the functional language in the former actor formalism. The state of an actor is however treated more explicitly in this algebra.

Language Syntax

The actor language uses behaviour definitions of the form $C \stackrel{\text{def}}{=} P$, where P is a program build from sequences of actor primitives and guarded choices. The actor primitives are:

$\texttt{send}(e_1, e_2)$	sends a message with the content described by e_2 to the actor described by e_1.
$\texttt{create}(a, C, e)$	creates a new actor with behaviour C and a state with the value of e. The actor name a will be used to denote the created actor in a following program sequence.
$\texttt{become}(C, e)$	changes the actor behaviour to C and sets the state to the value of e. The actor is freed to receive a new message and a new anonymous actor is created to complete the execution of a following program sequence.

A guarded choice has the form $e_1 : P_1 + \ldots + e_n : P_n$. Such a program will choose any branch P_i, which has a guard e_i that evaluates to true. Sequencing of actor primitives and programs is denoted by a ".".

The value expressions $e, e_1, \ldots$ in the above can make use of the special expressions `state`, `self` and `message`. The expression `state` accesses the state of the actor, the expression `self` refers to the name of the actor and the expression `message` holds the content of the last received message.

The following example illustrates how an object with a state and operations on the state could be modelled. The example assumes that a message has a field telling the kind of message, and that this field can be accessed using a "`.kind`" suffix. The recursive behaviour *Class* is defined. The actor *object* is created from this behaviour and a message is sent to invoke operation m_1:

$$\begin{array}{lcll} Class & \stackrel{\text{def}}{=} & \texttt{message.kind} = m_1 : & \ldots method_1 \ldots \texttt{become}(Class, m_1(\texttt{state}, arg)) \\ & + & \texttt{message.kind} = m_2 : & \ldots method_2 \ldots \texttt{become}(Class, m_2(\texttt{state}, arg)) \\ & + & \cdots & \cdots \\ & + & otherwise : & \texttt{send}(self, message).\texttt{become}(Class, \texttt{state}) \end{array}$$

$$\texttt{create}(object, Class, s_0).\texttt{send}(object, (m_1, arg))$$

A form of inheritance is added to the algebra by allowing a behaviour to be extended with new 'operations':

$$Class \stackrel{\text{def}}{=} e_1 : P_1 + e_2 : P_2 + \ldots + e_n : P_n$$
$$Class \stackrel{\text{ext}}{=} e_{n+1} : P_{n+1} + e_{n+2} : P_{n+2} + \ldots + e_{n+m} : P_{n+m}$$

A subsequent creation of an actor from the *Class* behaviour will result in the actor getting all $n + m$ branches.

Semantics

The semantics of the actor algebra is given in a way similar to that of the π-calculus. The operational semantics is given as a structural equivalence relation ($\equiv$) and a system of labelled transition relations ($\xrightarrow{\mu}$) between actor terms.

An actor term is a parallel composition of actors and messages, as defined by the following abstract syntax:

$$A ::= \; A|A \;\mid\; {}^{a}C_s \;\mid\; {}^{a}[P]_s \;\mid\; \langle a, v\rangle \;\mid\; A\backslash a \;\mid\; 0$$

${}^{a}C_s$ denotes an idle actor with behaviour C, name a and state s whereas ${}^{a}[P]_s$ denotes an active actor executing program P. $\langle a, v\rangle$ is a message in transit with receiver a and content v. $A\backslash a$ restricts the actor name a to be known only inside A. 0 denotes the empty actor. The restriction operator is used like in process calculi to specify the actors that know a specific name. When an actor name is sent from one term, where the name is restricted, to another term where the actor name isn't known, the restriction will be extended to cover both terms.

The labels in the labelled transition system specify the communication with the environment. The possible labels are:

γ	an invisible internal action
av	the receiving of a message with receiver a and content v
$\overline{av}L$	the emission of a message with receiver a and content v. The set L contains those actor names in v that have been sent out of their scope.

These labels are similar to the labels in the labelled transition system for the π-calculus. The set of labelled transition relations is specified by a set of axioms and rules, which also closely corresponds to the rules from the π-calculus. The rules *Out*, *In* and *Com* from the π-calculus correspond to the following *Deliver*, *Receive* and *Sinc* rules:

Deliver	$\langle a, v\rangle \xrightarrow{\bar{a}v} 0$
Receive	${}^{a}C_s \xrightarrow{av} {}^{a}[P\{v/\texttt{message}\}]_s \quad \text{if } C \stackrel{\text{def}}{=} P$
Sinc	$\dfrac{A \xrightarrow{av} A' \quad B \xrightarrow{\overline{av}L} B'}{A\|B \xrightarrow{\tau} (A'\|B')\backslash L}$
Become	${}^{a}[\texttt{become}(C, e).P]_s \xrightarrow{\tau} ({}^{d}[P\{a/self\}]_s)\backslash d \;\mid\; {}^{a}C_{[\![e]\!]^a_s} \quad d \text{ fresh}$

The last rule for the actor primitive `become` expresses that an anonymous actor is created to complete the execution of the program. This is just like the approach in the functional actor formalism.

The theory of observation equivalence from the π-calculus can be applied directly to this actor algebra because the semantics is based on the same kind of labelled transition system.

4.3.4 Actor Languages

Many actor based programming languages [Kim97][Rad99][VA01] add to the actor model to extend the expressiveness. This includes explicit notions of location and migration, synchronization constraints, and many new communication abstractions. Such special features of several actor languages will be described in the following.

THAL

THAL (Tailored High-level Actor Language)[Kim97] provides several communication mechanisms, uses synchronization constraints, and adds a migration mechanism that allows the locality of actors to be controlled.

A THAL program consists of behaviour definitions from where new actors can be created. A behaviour defines acquaintance variables (the state) and operations. The state is changed incrementally by updating the acquaintance variables. The sending of a message (invocation of an operation) is indicated by an arrow, and has the form:

`actorReference <- operationName(` $\texttt{arg}_1$`,` $\texttt{arg}_2$`,...,` $\texttt{arg}_n$`)`

Operation names are first-class values that can be assigned to variables or sent in messages.

Operation calls with a return value is supported by the concurrent call/return communication (CCRC) abstraction. It allows an actor to request the execution of an operation and receive a result in reply. This form of message passing is indicated by a "." instead of an arrow. The receiver of the request uses the `reply` construct to send back a result to the caller of the operation. CCRC allows the sender of a request to continue execution after it has sent the request. The sender doesn't block until it actually needs the result of the request. This allows an actor to send off several requests for concurrent execution at different actors. In the example:

```
sum = act1.m1() + act2.m2()
```

the second request doesn't need the result from the first request so the request will be sent as soon as the first request has been sent. However, the sum can't be calculated without the results, so the actor will now block until it has received both results.

Delegation of operation calls is made possible as a separate communication abstraction. The delegation of a message is indicated by a "!" instead of an arrow, and it has the effect of having the receiver send a reply directly to the reply destination of the sender, instead of to the sender itself. In the following example a client sends a message to a server and waits for a result. The server forwards the message to another server, which returns the result directly to the client:

client: `val=server.compute()`
server:
```
method compute()
      server2!compute()
end
```
server2:
```
method compute()
      reply result
end
```

Broadcast of messages is also supported in THAL by the use of groups. A group consists of homogeneous actors, which have the same method definitions. A group is created from a specified behaviour and group size, and it gets a unique name. A message sent to this group name will be send to all members of the group.

THAL supports local synchronization constraints, which allows one to specify which messages can be accepted in the current state. The constraints are disabling constraints and they are specified separately for each operation using the `restrain` expression:

`restrain` *operationName(arg_1, arg_2,..., arg_n)* `with (` *bool-expr* `);`

bool-expr is a boolean expression over the operation arguments and the acquaintance variables.

THAL systems can execute on several processors and it provides a way of specifying on which processor a particular actor should run. When a new actor is created in THAL, a location (processor identifier) where the actor should be created may be specified. Once an actor has been created it may be moved to another processor by sending it a `migrate` message with a specified location. However the behaviour definition must be available at the destination.

SALSA

SALSA (Simple Actor Language, System and Architecture) [VA01] is another actor based programming language with explicit support for migration. It also introduces token-passing continuations, join continuations and first-class continuations to coordinate the interaction between actors.

SALSA programs contain behaviour definitions, which can be instantiated to create actors. The behaviours define instance variables and operations, which can be invoked by the receipt of a message. The sending of a message is indicated by an arrow, and has the form:

`actorReference <- operationName(` $\texttt{arg}_1$, $\texttt{arg}_2$,..., $\texttt{arg}_n$`)`

Salsa uses a continuation passing style to support delegation, message passing where a reply is returned, and other forms of coordination of the message passing.

The passing of a return value is supported by token-passing continuations. A token-passing continuation specifies the actor, to whom the value (the token) should be returned, the kind of message (operation) that the value should be returned in, and the position of the value in the arguments of that message. The token-passing continuation mechanism is more general than the concurrent call/return communication in THAL. This is because it can not only be used to specify that a reply should be returned to the sender of a message, but also that a reply should be sent to someone else. On the other hand, it can't specify that the sender of a message should block until a reply is returned.

The token-passing continuation in the following example makes the server return a result in a `useResult`-message to the client:

```
client:  server<-compute()
             @ client<-useResult(token,otherArg)
```

`token` is a special keyword which represents the return value. Several token-passing continuations may be used in sequence.

A variant of the token-passing continuation is the join continuation. It allows a client to receive return values from several method calls in a single message. In the following example, a client sends messages to two servers and then receives both return values in an array in a `receiveResults` message:

```
client:  join(server1<-m1(),server2<-m2())
           @ client<-receiveResults(token)
```

First-class continuations support delegation of messages to third parties, allowing a result to be returned directly to the client. It makes use of the keyword `currentContinuation`. In the following example a client sends a message to a server and waits for a result. The server forwards the message to another server, which returns the result directly to the client:

```
client:  server<-compute() @ client<-receiveResult(token)
server:  int compute(){
                server2<-compute() @ currentContinuation
         }
```

SALSA supports distributed actor systems where actors run in different hosting environments, called theatres, and still communicate as though it was local communication. An actor can migrate between the theatres by using the `migrate` operation that is part of every actor. Since the migration support isn't obtained through a special primitive but through an ordinary operation, the migration can be triggered from both the actor itself and from other actors. The behaviour definition for the migrating actor has to be present at the receiving theatre, so only the state of the actor is actually being moved.

ABCL

ABCL (Actor-Based Concurrent Language) differs from the actor model by preserving the order of the messages. Further more it has three types of message passing mechanisms: *past* (asynchronous), *now* (synchronous), and *future* (eager).

Past is the usual non-blocking message passing with the addition that a reply destination may be specified, i.e. to whom a reply should be sent. *Now* is a blocking message passing where the sender waits for a reply, i.e. it simulates operation calls. *Future* is a non-blocking message passing where a reply is expected to arrive at some point of time in the future.

Concurrent Aggregates

Concurrent Aggregates differ from the actor model by using two-way communication as the default communication mechanism instead of the one-way message passing. The sender of a message is by default expecting a reply.

Concurrent Aggregates also provide one-to-one-of-many communication. This is obtained by grouping actors of the same kind into named aggregates. A message sent to an aggregate will be received by only one of the constituents. However, exactly which constituent that will receive the message is left unspecified.

4.4 Ambient Calculus

The ambient calculus [CG01] is a calculus concerned with the interaction of processes and their movement through different domains or locations.

Ambients are bounded places containing processes and data. They can be used to describe both computations and computation environments and particularly the movement of such. Among the concepts that ambients can model are numbers, data cells, locks, firewalls, π-calculus channels and mobile processes.

The security aspect is important in the ambient calculus. An ambient should not only have authorization when it needs to run or to access information in certain domains; it should also have authorization when it wants to enter or exit a domain. E.g. to cross firewalls or to move from a local computer to a LAN and on to a WAN.

4.4.1 Mobile Ambients

An ambient can be seen as a local computation environment, containing both data, code and processes. An ambient may also contain other ambients, giving a hierarchic structure of computation environments.

Ambients can move under the control of their contained processes. A process may guide its surrounding ambient to move up or down the hierarchy of ambients. An ambient can be moved out of its surrounding ambient or into a sibling ambient, or it can be dissolved to reveal its content.

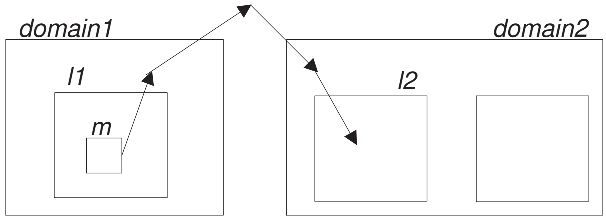

Figure 4.3: The two locations l_1 and l_2 are in different administrative domains. As a result, the ambient m in l_1 will have to use four steps to move to l_2.

An ambient is a bounded place, and everything inside the boundary move as a unit. So when an ambient is moved, all its contained processes and all its sub-ambients will move too.

Each ambient has a name; however it need not be unique. Knowing the name of an ambient gives the capability to enter the ambient, to leave the ambient or to dissolve the ambient. So the name of an ambient should be guarded closely and not handed out to every process. However it is possible to hand out a particular capability, e.g. the ability to leave an ambient, without revealing the name of the ambient.

For an ambient to be moved by one of its contained processes, the process needs to have the capability to leave the enclosing ambient or the capability to enter a sibling ambient. These capabilities correspond to authorizations to leave or enter administrative domains.

An ambient can not move directly into any other ambient (location) but will have to move in steps. An ambient can only change its position in the ambient hierarchy one level at a time. Consider the case in figure 4.3: A component m in location l_1 wants to move to location l_2 in another administrative domain. This can only be done by first moving out of l_1, out of the current administrative domain, into the other domain and finally into l_2. So an ambient can't do with just knowing the name of the destination location; it will also need to know how to get there. Also, since an ambient can't move to its destination location in one atomic action, other processes inside the ambient may run while the ambient is on the move.

Processes inside ambients can communicate using asynchronous message passing. The ambient calculus only allows processes inside the same ambient to communicate. This is because a message can't just travel between different administrative domains but will have to cross firewalls. However, message transport between ambients can be modelled by wrapping the message in an ambient that has the necessary capabilities to travel between the other ambients.

A process can output a message containing a name or a capability into the local ambient whereafter a process can pick up the message. The communication is anonymous, i.e. the sender doesn't know who will receive the message and a receiver doesn't know who sent the message. The asynchronous output can be used to store values for later retrieval. Since a message don't have a name, it can be wrapped in an ambient whose name can be used to denote the value.

4.4.2 Syntax and Semantics

The terms of the ambient calculus are processes P and capabilities M. They are described by the following abstract syntax:

$$\begin{array}{ll} P ::= & (\nu n)P \mid 0 \mid P|Q \mid !P \mid M[\,P\,] \mid M.P \mid (x).P \mid \langle M \rangle \\ M ::= & \texttt{in}\ M \mid \texttt{out}\ M \mid \texttt{open}\ M \mid x \mid n \mid \varepsilon \mid M.M' \end{array}$$

The restriction $(\nu n)P$ creates a new name n with the scope of P. 0 is the inactive process. $P|Q$ denotes parallel composition of the processes. Parallel composition is associative and commutative. $M[\,P\,]$ is an ambient with the name given by M. $!P$ denotes replication and $M.P$ executes the action denoted by the capability M. $\langle M \rangle$ denotes the asynchronous sending of capability M and $(x).P$ denotes the receipt of a message.

The formal semantics is given as a structural equivalence relation and a reduction relation ($\longrightarrow$) between processes. The rules for the reduction relation are given such that processes can evolve under restriction operators, under parallel composition and inside ambients. Besides such rules, there are rules that describe what happens when a process has a certain capability. The action depends on the kind of capability, as given by the rules below.

Entry capability: An ambient can be moved by one of its contained processes into a sibling ambient (figure 4.4 (a)):

$$n[\,\texttt{in}\ m.P|Q\,] \mid m[\,R\,] \longrightarrow m[\,n[\,P|Q\,] \mid R\,]$$

Exit capability: An ambient can be moved out of its surrounding ambient (figure 4.4 (b)):

$$m[\,n[\,\texttt{out}\ m.P|Q\,] \mid R\,] \longrightarrow n[\,P|Q\,] \mid m[\,R\,]$$

Open capability: A process can dissolve a sibling ambient to reveal the process inside (figure 4.4 (c)):

$$\texttt{open}\ n.P \mid n[\,Q\,] \longrightarrow P|Q$$

If no ambient of name n can be found, the process will block until such an ambient exists. Likewise for the other capabilities: they will block until correctly named ambients appear.

Message passing is described by the following rule, where a process picks up a pending message:

$$(x).P|\langle M \rangle \longrightarrow P\{M/x\}$$

The communicated value M can be a name, a capability, or a path of capabilities. ε denotes the empty path. As can be seen from the rule, there is no way to differ between messages; if there are several pending messages, anyone can be picked up independent of the content.

An observational equivalence between processes can be defined for the ambient calculus. The equivalence is based on the ambients that show up on the top level of the

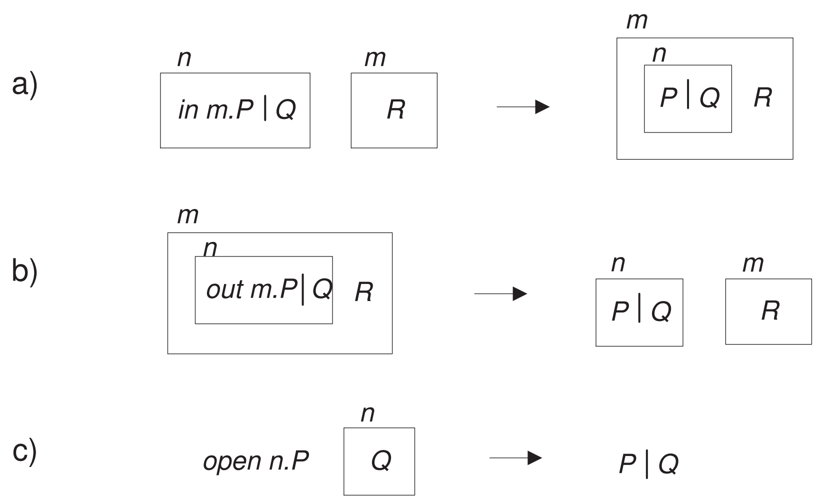

Figure 4.4: An ambient can move into a sibling ambient (a), out of a surrounding ambient (b), or be dissolved (c).

hierarchy of ambients, because such top-level ambients with unrestricted names are what can be observed from the environment. Such ambients may contain messages and they can be seen as messengers to the environment. The observation equivalence is thus based on the communication with the environment, just like the observation equivalences defined for the other formalisms. For the ambient calculus, two processes are said to be equivalent if they exhibit the same top-level ambients during computation.

Chapter 5

Analysis of the Formal Languages

The formalisms presented in chapter 4 have both differences and similarities. Processes are a basic concept in all of the formalisms, but the way they are organised and the way they interact are different. Message passing is the primary communication form in all of the formalisms, but it takes different forms and this gives different ways of synchronizing the processes. Further more some of the formalisms have an explicit location concept which eases the modelling of code and process mobility.

In the following, the formal languages will be analysed with respect to the requirements proposed in chapter 3. This will give a clearer picture of the differences between the formalisms and of the language constructs necessary to model systems with mobile code.

For each formal language, it will be discussed whether and how it can model the required aspects. The grouping of the requirements used in chapter 3 will also be used when analysing the languages with respect to these requirements. A full listing of the requirements can be seen in appendix A and a summary of whether the formalisms fulfil the requirements can be found in appendix B.

5.1 π-Calculus

Components: The polyadic π-calculus has two kinds of components: processes and process abstractions. A process can be seen as an executing component whereas a process abstraction can be conceived as a code component from where several concrete processes can be created. A passive component can be modelled as a collection of processes which are all just waiting to receive a message and won't start executing until the process receives a message requesting its execution. A state can be saved in messages on restricted channels.

Processes can be created dynamically and be composed to a common system using parallel composition.

Locations: There is no explicit location concept in the π-calculus. Instead, the locality of a process is modelled through its communication links to other processes. Special processes can be used to represent each location and can take care of creating links between processes within the location.

Mobility: In the monadic and polyadic π-calculus, the movement of a process is modelled as the changing of the communication paths. There are thus no explicit primitives for modelling the movement of a process to another location and it is up to the modeller to control which channels should be created, changed or removed to simulate the movement of a process.

In the higher order π-calculus, processes and abstractions can be sent between other processes that represent different locations. A moved process can start executing at the new location when the receiving process decides to. A moved abstraction can be used to create new executing components.

There is no distinction between whether a process is being shipped or fetched since there has to be both a sender and a receiver. To model that a process moves at its own will, the process simply sends its own continuation and it thus doesn't differ from the situation where a process is being fetched or shipped. The movement of a process is asynchronous like every other communication, but synchronous moves can be modelled by having the moved process send a signal on a specific channel when it is about to finish its execution.

To model that a copy of a process is being sent, the process has to be held in a process variable, which can still be used after the process has been sent.

Bindings: The process or abstraction that is being sent is the unit of mobility; there are no references to other components that should move too. All known channel names will still be valid after the movement; if a communication link should become invalid, it is up to the modeller to stop using the corresponding channels.

Security: When locations are modelled using special processes to represent the locations, a moving process will have to contact the involved location processes to establish the proper connections. A process can be kept from moving to a certain location if it doesn't know the name of the channel where the location process is listening. Besides this, there is no way to restrict processes.

Openness: The π-calculus allows new channel names to be created and communicated to other processes, such that new communication links can be established.

Communication: Communication through channels allows both one-to-one and one-to-one-of-many communication, depending on how many knows the channel. In the presented version, the communication is asynchronous but synchronous communication can be modelled using some sort of meeting protocol. In a synchronous π-calculus, asynchronous communication can also be modelled.

The communication is inherently one-way, but two-way communication can also be modelled. The invocation of an operation can be modelled as the sending of the arguments on a particular channel. The name of a reply channel should also be sent, such that a result can be returned to the process:

To invoke operation f: $(\nu\kappa)\ f\langle\tilde{v}, \kappa\rangle \mid \kappa(x).P$
The operation f: $!f(\tilde{x}, \kappa) \ldots \kappa\langle f(\tilde{x})\rangle$

The addressing of the one to communicate with is location independent.

Synchronization: The blocking input action can be used to obtain synchronization. Input from and output to a lock channel can be used to obtain mutual exclusion between processes. This is illustrated with the state channel in the object example below. Condition synchronization can be modelled by signalling on a particular channel when the condition is fulfilled.

Failure Handling: Failure in a component can be modelled by letting the corresponding process enter an error state where it doesn't respond to any messages. It will however not be possible for other processes to detect this error; they will wait for a response forever.

Object Properties: Objects can be modelled in the π-calculus by using a restricted channel to save state information in, and by introducing new channels to represent the operations.

The state of an object can be stored in a message on a particular channel. The channel should be restricted to be known only in the processes that model the operations of the object. This provides the encapsulation of data.

An operation can be modelled as replicated input from a particular channel, allowing a process to keep accepting messages on that channel. The process will read the state from the internal channel and will output a new state. The channel names that are introduced for the operations should be returned to the creator of the object, to allow the operations to be used from other processes. This allows one to keep some operations private.

An abstraction can be used to define a class, which can be instantiated to objects. The example below illustrates the idea. The abstraction takes a formal parameter which is the channel on which the operation names should be returned. The abstraction is used to create an object, and one of the operations is invoked.

$$
\begin{array}{lll}
Class \stackrel{\mathrm{def}}{=} (ch) & (\nu m_1)(\nu m_2)\ldots(\nu m_n)(\\
& ch\langle m_1\ m_2 \ldots m_n\rangle \\
& \mid (\nu\ state)(\\
& \quad state\langle s_0\rangle \\
& \quad \mid\ !m_1(arg)\ldots state(s)\ldots(state\langle m_1(arg,s)\rangle \mid \ldots) \\
& \quad \mid\ !m_2(arg)\ldots state(s)\ldots(state\langle m_2(arg,s)\rangle \mid \ldots) \\
& \quad\ \ \vdots \\
&) \\
\qquad\qquad) & \\
(\nu\ object)(& Class\langle object\rangle \\
& \mid object(m_1\ m_2 \ldots m_n).m_i\langle arg\rangle \\
\qquad\qquad) &
\end{array}
$$

An abstraction can inherit from another abstraction simply by using the abstraction constant to create a concrete process in parallel with some extra processes:

$$
\begin{array}{l}
C_1 \stackrel{\mathrm{def}}{=} (\tilde{x})P_1 \\
C_2 \stackrel{\mathrm{def}}{=} (\tilde{x})\ C_1\langle\tilde{x}\rangle \mid P_2
\end{array}
$$

However, it is a limited form of inheritance because the extension P_2 can't use any of the private channels of P_1.

5.2 Nomadic π

Components: Nomadic π has the same components as the π-calculus but it adds agents. An agent is a composition of processes into a unit and as such is no different from the composition of executing components in a system; it only serves to define which processes should migrate together.

Locations: The concept of sites directly models locations. Agents and processes that initially are located at different sites are specified separately whereas agents and processes in the same location are specified together. In the Nomadic Pict programming language separate programs are written to be executed at each their site. The sites have a flat structure.

Mobility: The movement of executing components is directly modelled as the migration of agents. All processes in the body of an agent move together and continue execution where they left, upon arrival at the new location. State information output to internal channels will migrate together with the agent, just like every other process in the body of the agent.

Agents move at their own will; shipping or fetching must be modelled by sending the agent a request to migrate. The destination location is specified by its unique site name.

The following example illustrates how a process can request the creation of an agent that starts by moving to a particular location. The name of the new agent is returned to the requesting process to allow communication with the agent.

```
agent a = (
  *create?(s r) → agent b =
                      (migrate to s → ...)
                  in r!b
  | new agentch in (
        create![s1 agentch]
        | agentch?(agent) → ...
    )
  )
in ()
```

The above code can produce copies of identical agents at different locations. It is, however, not possible to model that a copy of an executing agent is created at a remote location.

Bindings: Bindings are treated similarly to the π-calculus.

Security: There is no way to model security mechanisms in Nomadic π. An agent can migrate to any location it knows the name of and perform any action. The site names are static names, and they can thus become known by any agent.

Openness: Nomadic π allows both site names, agent names and channel names to be transmitted on channels in order to allow new communication links to be created.

Communication: All communication, between processes within an agent or between different agents, goes through channels just as in the π-calculus. However, the receiving agent must also be specified when outputting to a channel and this makes one-to-one-of-many communication impossible.

The calculus supports location dependent as well as location independent addressing. A translation between the two must however be provided. This makes the calculus particularly useful for specifying the communication infrastructure of a system, e.g. how a middleware works.

Synchronization: The synchronization mechanisms are the same as in the π-calculus.

Failure Handling: As for the π-calculus.

Objects: The modelling of an object in Nomadic π is basically the same as in the π-calculus. However, agents now provide a natural way of modelling objects because they are the unit of mobility. They also provide a more well-known way to address the operations in an object: the name of an agent can be seen as the reference to the object and the channel name gives the operation. However, agents can't be extended in the same way as process abstractions.

Location

$$\underbrace{\{\!|\ \begin{array}{l} m_1\langle args\rangle|state\langle s\rangle \triangleright \ldots \quad , \\ m_2\langle args\rangle|state\langle s\rangle \triangleright \ldots \quad , \\ \ldots \end{array} |\!\}}_{code} \vdash \{\!|\ \underbrace{state\langle\cdot\rangle}_{state} \mid \underbrace{\ldots m_2\langle\cdot\rangle\ldots}_{control\ information}\ |\!\}$$

Figure 5.1: A location can be conceived as a component with a code part and a state part. The reduction definitions constitute the code, and the processes constitute the state and the control information for the execution.

5.3 Distributed Join-calculus

Components: The main component in the distributed join-calculus is that of a location. A location holds reduction definitions and a set of processes, which can be either pending messages, definitions of new reductions, or `go`⟨⟩, `halt`⟨⟩ and `fail`⟨⟩ primitives.

As described earlier, the reduction definitions can be seen as function declarations that define the behaviour of the location. The reduction definitions can thus be seen as the code component of a location. The same reduction definitions can however not be reused to create several locations, as is required from a code component.

The processes in a location can be seen as state information together with control information. A pending message on an internal port can represent the state (see the object example below). Other pending messages, definitions, or primitives represent ongoing computations and they mark how far the computation has got. This way of conceiving the different parts of a location as a code part and a state part is illustrated in figure 5.1.

Depending on the kind and number of processes in a location, the location can be conceived as a passive or executing component or even a component where several threads of execution take place at the same time. The pending messages that represent ongoing computations can be joined by a reduction definition to one thread of execution and later again be split in several threads of execution. Also, the computations can move from one location to another, depending on which location holds the needed definition. This is similar to an executing component that has references to other passive components and uses these in the computation. However, in the join-calculus the control information is distributed between the locations instead of being kept in one place.

The distributed join-calculus allows locations, processes and reduction definitions to be created dynamically and to be composed in a concurrently executing system.

Locations: A location in the distributed join-calculus can be understood both as a component and as a location, as the term is used in the requirement section. The locations are organized in a tree and the locations near the root of the tree can thus be conceived as physically separated locations whereas locations further down the tree can be conceived as components. There is however no distinction between the two forms of locations.

The hierarchic structure of the locations doesn't influence who can interact with who; the structure is only important when modelling failure and when determining the components that should migrate together.

Mobility: A location can migrate to another location, to become a sublocation of that one. Only a location as a whole can be moved, bringing all its definitions, processes and sublocations. When a location has arrived at the destination location, it will continue any ongoing computation. It is also possible to specify a continuation in the go-primitive, in order to start a process upon arrival. The destination location is specified by its name.

A location migrates at its own will; fetching and shipping should be modelled by having the location offering a migrate operation that can be invoked from other processes. The move is asynchronous, but synchronous moves can be modelled by the use of continuations.

It isn't possible to create a copy of a process or a location, except when the process or location is defined in a reduction (see the object example below).

Bindings: When a location migrates, all its sublocations will migrate too. This can be used to model that certain components should migrate together. Bindings to those components that don't migrate will still be valid.

Security: A location can only move to a location it knows the name of, and since fresh location names are generated for each location, a location's name will not be generally known. By guarding who a location name is given to, certain locations can be kept from moving there. Likewise with port names; a process can't send messages to a port unless it has been declared in the same scope as the port or has received the port name in a message. Besides this, there is no way to control the actions of a location or process once it has gained access to another location.

Openness: New channel names can be created and transmitted in messages to allow a reply to be returned to the right process and to otherwise allow dynamic creation of new communication links.

Communication: The communication builds on asynchronous message passing through channels, as is the case in the π-calculus. However, only one location can input from a channel, and the receiver of a message is thus uniquely identified by the channel name. This only allows one-to-one communication. The addressing of the receiver is location independent, as the port name uniquely identifies the receiver no matter its location.

The message passing is inherently one-way, but syntactic sugar is provided to support synchronous two-way communication.

Synchronization: Jointly defined functions provide the same synchronization capabilities as the blocking input action in the π-calculus[FG96]. A lock port can be used to model mutual exclusion, as is illustrated with the *state*-port in the object example below. Condition synchronization can be modelled in a similar way, by defining reductions with a join-pattern consisting of messages on several ports.

Failure Handling: The distributed join-calculus has explicit primitives for modelling failure in a location and for letting other locations detect that a failure has occurred.

Object Properties: Objects can be modelled using the same principles as for other process calculi. An object is modelled as a location with a reduction definition for each operator and an internal channel to hold the state. The join-calculus has no concept of a code component that can be used to create several passive components. Instead, a class is modelled as a function that defines a location. An object can be created by invoking the function:

$$
\begin{array}{lll}
\texttt{def } class\langle s_0\rangle \triangleright & \texttt{def } obj[& m_1\langle arg\rangle | state\langle s\rangle \triangleright \ldots state\langle m_1(s, arg)\rangle \\
 & & m_2\langle arg\rangle | state\langle s\rangle \triangleright \ldots state\langle m_2(s, arg)\rangle \\
 & & : state\langle s_0\rangle \\
 & &] \\
 & \texttt{in reply} & m_1, m_2 \texttt{ to } class \\
\multicolumn{3}{l}{\texttt{in let } m_1, m_2 = class\langle s_0\rangle \texttt{ in } m_1\langle arg\rangle}
\end{array}
$$

Operations can be kept private if the corresponding port names aren't transmitted out of the location.

5.4 Actors

Components: The actor model works with two kinds of components: behaviours and actors. A behaviour can be seen as the code component from where several actors can be created. An actor has a behaviour, a state, and an associated thread of control, and it can thus be seen as an executing component. However, an actor is reactive in the sense that it only starts executing in response to a message, and it can thus also be conceived as a passive component that is used by some other executing component.

An actor is defined from a new behaviour every time the actor becomes ready to receive a new message. This makes it necessary for the behaviour definition to be co-located with the actor.

Locations: The actor model has no explicit notion of location and there is no direct way to specify whether two actors are at the same location or at separate locations. However, the locality of an actor can be modelled through its acquaintance with other actors, and special actors can be used to represent different locations. Some actor languages add a "site" concept to support distributed computing.

Mobility: The actor model has no explicit primitives for modelling the movement of an actor. The movement can though be modelled as the changing of actor acquaintances, but it is up to the modeller to control this and give it the right interpretation.

An actor can create a copy of itself by using the actor primitive *create* to create a new actor with a similar behaviour and state. The new actor will not be executing from the same point in the program, but it is possible to use this method to model that a copy of an actor should be moved.

Bindings: When an actor moves, all its references to other actors will still be valid.

Security: An actor can be kept from moving to a certain location in the same way as in the π-calculus. An actor is given a new unique name when created and it is therefore possible to limit the actors that know this name and thus can interact with it. Otherwise it isn't possible to limit an actor's capabilities.

Openness: Actor names, and often also message kinds, can be sent in a message such that other actors can become acquainted with them and new communication links established in this way.

Communication: The communication is based on asynchronous one-way message passing. Unique actor names are used to specify the recipient of a message, and this makes the addressing location independent and provides one-to-one communication. Some actor languages allow actors to be grouped together to allow either one-to-many communication as in THAL, or one-to-one-of-many communication as in Concurrent Aggregates.

Sending a message is analogue to invoking an operation, but no returning of a reply is involved. Like in the process calculi this can be modelled with a continuation passing style. The name of the requesting actor (and possibly also the name of the message kind) can be included in the message with the operation request, and the operation will then send a reply to this actor.

However, it isn't possible to model that the requesting actor blocks until a reply is returned. This is because an actor only has one mailbox where all kinds of messages are received, so the actor can't wait for one specific message to arrive. In order to model this, the language will need some kind of guard or constraints on the message receipt. See the *synchronization* section below.

Most actor programming languages add some way of modelling two-way communication and additional communication mechanisms. E.g. the CCRC abstraction in THAL, token-passing continuations in SALSA, and the *Now* message passing in ABCL.

Synchronization: An actor has only one thread that can access the actor's state, so there is an implicit mutual exclusion. Other forms of synchronization are not possible with the basic actor model since there is no way of letting an actor block until a specific message arrives or a specific condition is satisfied. Many actor languages add a mechanism, which lets an actor sort the incoming messages, e.g. synchronization constraints, but such a mechanism isn't absolutely necessary. An actor waiting for a particular message can simply resend every other message to itself for later handling, like a busy-wait.

Different mechanisms have been used with the actor model: The actor language Rosette [Rad99] uses enabled sets, THAL uses synchronization constraints.

Enabled sets specify the message kinds that the actor is willing to receive. The first message in the mailbox, contained in the currently enabled set, will be the next message to be received. The enabled set should reflect the current state of the actor and should thus change every time the actor becomes ready to receive a new message.

Synchronization constraints are predicates that constrain the delivery of messages. The constraints should depend on the state of the actor. A message is delayed until it satisfies the constraints (enabling constraints). Disabling constraints can also be used, i.e. a message satisfying the constraint will not be accepted.

Failure Handling: Failure in an actor can be modelled by letting the actor have a behaviour where it will never execute a *become*. This will prevent the actor from receiving further messages. This can however not be detected from other actors because they can still send messages to the actors mailbox.

Object Properties: An object can be modelled directly as an actor, as shown with both actor formalisms. However, it isn't possible to model that some operations are private since everyone can use the general operation names. The use of general non-unique operation names will on the other hand allow polymorphism, because several actors can offer the same operations. A form of inheritance is supported by the process algebraic actor formalism but it isn't part of the general actor model.

5.5 Ambient Calculus

Components: The only components in the ambient calculus are ambients, which can contain processes, pending messages and other ambients. An ambient doesn't have just one thread of control but can have zero, one or more processes running on the top-level or in sub-ambients. The control information will be distributed between all the processes. A state can be stored in pending messages and sub-ambients. Because of the possibility of many running processes, an ambient can't be characterised as either a passive or an executing component but it can model both.

Locations: An ambient can not only be considered as a component but also as a location. The hierarchic structure of ambients allows the modelling of administrative domains with sub-domains and sub-sub-locations.

Mobility: The unit of mobility is an ambient, which moves together with all its contained processes and sub-ambients, i.e. only a component as a whole can be moved. The destination location is specified through the path of capabilities that is needed to get to the destination ambient. Several ambients can be named the same, so the destination may not be deterministic. When the ambient reaches the destination, it can continue executing a process or it can be manipulated by other processes.

An ambient can only be moved by one of its top-level processes, i.e. an ambient moves at its own will. However, an ambient can allow other ambients to enter it and be dissolved in order to create a new top-level process, which can then move the ambient. This can model the shipping or fetching of an ambient. The movement of an ambient is asynchronous, but synchronous moves can be modelled by using an ambient as a messenger to tell that the computation has finished.

Bindings: When an ambient moves, it moves with all its sub-ambients and processes. This allows some components to move together with the moving component. The moving ambient can have capabilities that allow it to interact with other ambients in the environment. When the ambient is moved, the path to these other ambients will change such that interaction no longer is possible. However the ambient can be given new capabilities to allow further interaction. This allows the modelling of both bindings that are kept across the network and bindings that become invalid.

The ambient calculus also allows the modelling of by-type bindings. Several ambients can be named the same, and this allows a similarly named resource to be present at every location. An ambient with the capability to interact with such a resource will be able to use the same capability to interact with a similar resource at another location.

Security: Locations are not uniformly accessible, as they are in all the other formalisms. A component need authorization to leave or enter a domain, and the access to a location can be controlled by only handing out entry capabilities to trusted components.

The ambient calculus allows a component to be restricted in other ways. A message cannot be sent to any other ambient, unless the sender has the necessary capabilities to travel there. Likewise, the message cannot be picked up by any ambient, unless it has the capabilities to enter or dissolve the ambient that contains the message.

An ambient cannot be moved by just any process. For a process to move an ambient it has to be a top-level process inside the ambient. If the process isn't already there, it has to first get the capability to enter the ambient and then another process has to open up its surrounding ambient.

Openness: New names can be generated and names and capabilities can be sent in messages to allow an ambient, and thus messages, to be able to move to new locations. This can be used to establish new communication links.

Communication: The ambient calculus uses asynchronous message passing for communication between processes. A message can be picked up by any process inside the same ambient as the pending message. A message can also be directed to a specific ambient by packing the message inside an ambient, which can transport the message to the proper receiver. This can model both one-to-one and one-to-one-of-many communication.

The communication is one-way but two-way communication can be modelled by also sending the path necessary to return a result. The example below illustrates how the request of an operation and the return of a result can be modelled. A client invokes an operation f in a server ambient and waits for a reply to be returned. The operation request is modelled as an ambient with the name of the operation. The ambient contains the arguments of the operation request and a process that first guides the ambient to the server and then guides a reply back to the client. Another process in the client will wait for the reply to be returned:

$$\begin{array}{lll} \textit{client} & [& f\ [\ \langle \textit{args} \rangle \\ & & \quad\ \mid \texttt{out}\ \textit{client}.\texttt{in}\ \textit{server}.\textit{reply}[\ \texttt{open}\ \textit{return}.\texttt{out}\ \textit{server}.\texttt{in}\ \textit{client}\ ]\ ] \\ & \mid & \texttt{open}\ \textit{reply}.(\textit{res}).P\ ] \\ \textit{server} & [& !\texttt{open}\ f.(\textit{args})\ldots.\textit{return}[\ \langle \textit{res} \rangle \mid \texttt{in}\ \textit{reply}\ ]\ ] \end{array}$$

The addressing of the receiver of a message is location dependent since the sender has to know the path to the receiving ambient. Communication between processes within the same ambient is always possible whereas communication to other ambients can be restricted by not giving out capabilities that allow a message ambient to travel there.

Synchronization: Mutual exclusion and condition synchronization can be modelled using the possibly blocking `open` primitive. An empty ambient can be used as a lock, $\textit{lock}[\ ]$. A process whishing to obtain the lock, executes `open` *lock* to dissolve the ambient. When the process is done needing the lock, it releases the lock again, $\textit{lock}[\ ] \mid P$.

Failure Handling: The ambient calculus can't model complete failure in a component, because even though all processes inside the component stop executing, other ambients can still enter or exit the component.

Object Properties: An object can be modelled as an ambient that accepts operation requests, as shown in the client-server example above. A state can be stored in a message in a sub-ambient which name is only known within the object. Likewise, operations can be made private by restricting the operation names to be known only within the object.

A class can be modelled as an operation that creates an object ambient with a new name and returns an "object reference". The object reference would consist of the capabilities to enter and leave the object ambient:

$$
\begin{array}{l}
\texttt{!open}\ class.(s_0). \\
\ (\nu\ obj)(\ obj\ [\\
\qquad (\nu\ state)(\ state[\ \langle s_0\rangle \mid \texttt{!open}\ enter\ ] \mid \texttt{!open}\ exit \\
\qquad\qquad \mid \texttt{!open}\ m_1.(args)\ldots enter[\texttt{in}\ state.(x).(\ \langle m_1(s, args)\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid exit[\ \texttt{out}\ state\ldots]\ )\ ] \\
\qquad\qquad \mid \texttt{!open}\ m_2.(args)\ldots \\
\qquad\qquad \mid \cdots) \\
\qquad] \\
\qquad \mid return[\ \langle \texttt{in}\ obj\rangle\ \mid\ \langle \texttt{out}\ obj\rangle\ \mid\ \texttt{in}\ reply\ ]\ )
\end{array}
$$

As can be seen in the example, the code needed to modify the state gets rather complicated.

5.6 Comparison of the Languages

As seen in this chapter and by the summary in appendix B none of the existing languages fulfil all the requirements. None of the languages can model the mechanism used in RMI where the state of a component and the code component can move separately. And none of the languages can model that a resource or component only is accessible from within its current location.

However, the formalisms all have some basic constructs that allow them to model mobility and to fulfil most of the requirements. The languages use different constructs and have different approaches to how the required aspects should be modelled. This allows different requirements to be fulfilled. The following sections will compare the different approaches and look at their limitations and advantages. This will end up with an identification of the best approaches to the modelling of the different aspects.

5.6.1 Location and Component Constructs

Nomadic π has constructs both for modelling locations and for forming components: Processes and messages can be grouped into 'agents' and a separate 'site' concept is used to represent locations. The distributed join-calculus and the ambient calculus have only one construct that is used to model both locations and components. The distributed join-calculus operates with 'locations' and the ambient calculus operates with 'ambients'.

In the π-calculus and in the actor model the location of a component is modelled through its acquaintance with other processes. There is no explicit location concept. The actor model has a component concept, an actor, which encapsulates a state and a single process, but the π-calculus doesn't even have that. Processes can be composed in parallel but there is no construct that can separate the processes into components.

The existence of explicit constructs for the modelling of locations, components, and the migration of components between locations, make the modelling of these concepts much easier. The semantics of the language defines what it means to move a component, what happens to bindings when a component is moved, and how much of the state and how many processes that should be moved. Without explicit location and migration constructs, all this will have to be handled by the modeller who should control which communication links should be created and which should not be used any more. This, on the other hand, gives more flexibility and allows different migration forms to be modelled.

When the same construct can be used to model locations and components, it is because the construct simply creates a boundary around processes, messages, or other bounded components. This possibility of having boundaries nested inside each other, allows one to model nested components and nested locations, e.g. a network domain may be split up in several computers, which may contain several resources or programs, which again consist of different objects. This nesting of locations/components can be used to control that some components migrate together because a component migrates with all its subcomponents.

Using the same construct to model locations and components also allows one to use the same migration primitive to move components as well as locations (e.g. a computer that should be moved to another domain). On the other hand, it also leaves it to the modeller to control what should be understood as a component and what should be understood as a location.

When the same concept is used to model both locations and components, the mechanism that takes care of limiting the scope of the component names will also do this for the location names. This can be used to restrict the access to both a location and a component. This is not only the case in the ambient calculus and the distributed join-calculus but also in the π-calculus and the actor model. When a process or an actor is used to represent a location, other processes or actors have to know the channel name or the actor name of the location before they can move there, and this can be restricted similarly.

5.6.2 Naming of Components

The component constructs in the formalisms will, besides making a boundary, also assign a name to the component. The agents in Nomadic π, the locations in the distributed join-calculus, and the actors will all be given a unique name, which makes it possible to identify the specific component.

The location names in the distributed join-calculus are used when a location wants to move to another specific location. The agent names in Nomadic π and the actor names are used for addressing purposes when sending messages; the channel name or the message kind alone is not enough to uniquely determine the receiver of a message, and the component name is therefore also needed.

Ambients are also given names, but not necessarily unique names. The ambient names are the only names in the ambient calculus so they are used whenever an ambient moves or messages are transported between ambients. Several ambients can have the same name and this can give non-determinism but it also allows modelling of by-type bindings. However, this is only possible because the addressing of an ambient is location dependent, so it will be an ambient in a certain relative location that will be addressed and not a similarly named component somewhere else in the ambient hierarchy.

Besides operating with named passive or executing components, the polyadic π-calculus and the actor model also operates with named code components (process abstractions and actor behaviours). This makes it possible to reuse a code component because several passive components can get their own copy of the code simply by referring to the name of the code component. This can not only be used to construct several passive components from the same code component but also be used to support inheritance. In the polyadic π-calculus, where there is no component boundary, a process abstraction can be used inside another process abstraction, which in this way can inherit the processes from the first abstraction and add its own processes.

5.6.3 Addressing of Messages

As already mentioned, Nomadic π and the actor model use component names when specifying the recipient of a message. This is enough to identify the receiving component, but it is not enough to determine how the message should be treated. The actor model operates with different message kinds to allow the receiving actor to behave differently depending on the received message. Several actors can receive the same kinds of messages and this allows polymorphism; an operation can be written to work on any actor that accepts certain kinds of messages. This isn't possible when operations in a component are addressed through unique channel names that no other component can use.

The π-calculus, Nomadic π and the distributed join-calculus use channels for communication. A process can send a message to a channel and another process can pick up a message from the channel. The receiver is uniquely determined in the distributed join-calculus and in Nomadic π, whereas the π-calculus allows several processes to receive messages from the same channel. The last approach allows one-to-one-of-many communication.

The location dependent addressing form used in the ambient calculus allows different kinds of bindings to be modelled, but it also makes it difficult to model message passing between components or a simple update of a state variable; it all takes a lot of instructions.

5.6.4 The Ideal Constructs for a New Modelling Language

As already discussed, none of the described formal languages fulfil all the requirements and a new modelling language will therefore be developed in the following chapter. However, the described formal languages all provide some basic techniques or constructs for modelling concurrent components, mobility and communication (as described above) and this will be used when trying to construct the new modelling language. The following summarizes which language constructs have which advantages:

- Explicit language constructs for modelling locations, components and migration of components will ease the task of the modeller.
- Mobility modelled as the change of communication paths will give more flexibility.
- The nested location concept of the ambient calculus and the distributed join-calculus allows nested components and locations and it can model that some components should migrate together. It also allows access control to locations because fresh names can be generated for locations.
- A named code component allows process/behaviour definitions to be reused in several passive components and can also be used to support inheritance.
- The addressing form used in the actor model uniquely determines the receiver of a message and at the same time allows for polymorphism.
- The addressing form of the π-calculus allows for one-to-one-of-many communication.
- The location dependent addressing form in the ambient calculus combined with non-unique component names allow for by-type bindings.

One of the existing formalisms will be chosen as the starting point for a new modelling language. This choice will determine the style and syntax of the new modelling language. The language can then be modified and extended to allow further requirements to be fulfilled.

It will be prioritized that the new modelling language is easy to use, and the language should thus provide explicit constructs for locations, components and migration. A construct that allows nested locations and components has a lot of advantages and this should thus also be part of the new modelling language.

The ambient calculus has a nested location construct and further fulfils a lot of the requirements. However, it is also rather difficult to use and a lot of instructions are necessary when a message should be sent, an operation invoked, or a state variable updated.

The distributed join-calculus also has a nested location construct. The language provides primitives for failure handling, and the way objects and operation requests are modelled is more intuitive than in the ambient calculus. Further more, the location construct already provides a natural division of a component into a code part and a state part. The language just needs to be extended to allow the code part to be defined separately and be reused.

The distributed join-calculus is therefore chosen as the basis for the new modelling language. The construction of the new language will be described in the following chapter.

Chapter 6

Construction of a New Modelling Language

A new modelling language is needed if it should be possible to model the mobility mechanisms used in Java. As concluded in chapter 5, the distributed join-calculus is a good starting point for a new, easy-to-use, modelling language. The distributed join-calculus will therefore be modified and extended in the following in order to construct a language that allows more of the required aspects to be modelled.

The distributed join-calculus as presented in [FGL$^+$96] is taken as the basis. The following chapter will often refer to rules and definitions from the distributed join-calculus, and the definitions from the calculus is therefore summarised in appendix C. In order to simplify the introduction of new language constructs, the failure handling part of the distributed join-calculus will be left out in this chapter.

The construction of the new modelling language will be presented step by step in the following. Some of the existing rules from the DRCHAM will be modified and new rules will be added. The final constructs and definitions in the new language will not be presented formally until chapter 7.

6.1 The New Basic Language Constructs

The basic extensions and modifications, which allow code to be reused and code and state to be moved separately, will be presented below. The presented rules will not be the final rules but will only give the general idea of the working of the new constructs and will lay the foundation for the new language.

6.1.1 Reuse of Code

In order to allow code reuse, the language should be extended with named code components and the possibility of using these names to create passive components.

A code component in the distributed join-calculus is simply the definition part of a location. A location on the form $c[D : \mathtt{0}]$ can thus be seen as a named code component. It should be possible to use the name of such a location to create other locations. This can be done by writing the name, c, in the definition part of a new location, $a[c : P]$, meaning that location a should have the same definitions as location c.

Using this way of writing, a class definition and its instantiation to an object can be modelled in the following way:

$$\begin{array}{rll} \texttt{def} & \mathit{class}[& m_1\langle \mathit{args}\rangle|\mathit{state}\langle s\rangle \triangleright \ldots \quad \wedge \\ & & m_2\langle \mathit{args}\rangle|\mathit{state}\langle s\rangle \triangleright \ldots \quad : \mathtt{0}] \\ \wedge & \mathit{object}[& \mathit{class} : \ \mathit{state}\langle s_0\rangle] \\ \texttt{in} & \mathtt{0} & \end{array}$$

This means that the location *object* has the same definitions as the location *class* and further has a pending message holding some initial data.

When a location in this way is defined from another location, it is really the definitions from the latter location that is being copied to the former location. Only the definitions are being copied, not the location boundary. This will allow inheritance to be modelled because the copied definitions can be extended with new definitions. Continuing the above example, the following will create a subclass class_2 of the original *class*:

$$\mathit{class}_2[\mathit{class} \ \wedge \ m_3\langle \mathit{args}\rangle|\mathit{state}\langle s\rangle \triangleright \ldots : \mathtt{0}]$$

The location class_2 will contain the same definitions as *class* and further have an extra reduction definition.

When the name of a location is written in the definition part of another location, it means that all the definitions from the former location are copied to the latter location. There is however no reason to restrict it to only being the definitions that are copied; also processes in the former location should be copied to the latter location. This will allow a copy to be made of any location, no matter whether the location represents a code component, a passive component or an executing component. A clone of the object from the former example can for example be created simply be writing $\mathit{clone}[\mathit{object} : \mathtt{0}]$, which will create a location *clone* with the same definitions and processes as the location *object*.

The original DRCHAM needs to be extended to support this copying of definitions and processes from one location to another. This is done by adding the following *Copy* heating rule:

$$a[c \wedge D_1 : P_1] \vdash_\phi \ \| \ c[D_2 : P_2] \vdash_\psi \ \rightharpoonup \ a[\,D_2 \wedge D_1 : P_2|P_1] \vdash_\phi \ \| \ c[D_2 : P_2] \vdash_\psi$$

This rule will allow a location name appearing in the definition part of another location to be expanded to the content of the former location. Or put in another way: The definitions and processes in location c will be copied to location a. The rule is a heating rule because it changes the form of a molecule (the definition of location a) to a form where new reductions may take place.

6.1.2 Addressing of Messages

When the definitions from one location are copied to another location, several locations will end up having the same reduction definitions. This means that a port name no longer will be enough to uniquely determine the receiver of a message. The addressing form will therefore be changed to the form used in the actor model and Nomadic π. The location name will be used to determine the receiving component, and the port name will be used to decide how the component should react in response to the message. This will also allow polymorphism.

The new addressing form means that a pending message henceforth will have the form $a.x\langle\tilde{v}\rangle$, where a is the receiving location and x is a port defined in that location. If a message should be sent to a port within the same location, no location prefix will be needed in front of the port name.

The chemical rule that allows communication between locations is changed to support this new addressing form:

$$Comm: \qquad \vdash_{\phi} a.x\langle\tilde{v}\rangle \parallel \vdash_{\psi a} \quad \rightarrow \quad \vdash_{\phi} \parallel \vdash_{\psi a} x\langle\tilde{v}\rangle$$

The rule will allow a pending message to be transported to its destination location, whereupon the location prefix will disappear from the message. This means that a pending message without a location prefix is aimed at a port within the current location. Only when a pending message is on a form without a location prefix, can it take part in a reduction, i.e. a message $a.x\langle\rangle$ will not be reduced at the current location even though the location defines a port of name x.

The distributed join-calculus supports sequencing by allowing the name of a continuation port to be sent in a message. In order to still support the use of continuations, message addresses on the form $a.x$ will be treated as first class values that can be sent in a message. When a message containing a value of this form is matched against a join pattern, it will be matched against a single variable name (κ), which can then be used to create a pending message ($\kappa\langle\cdot\rangle$). Join patterns will not be allowed to contain patterns of the form $a.x$.

The syntactic sugar that is added to the join-calculus to support implicit continuations can still be used. The only change is that the name of a continuation channel always will be prefixed with a location name when the continuation is sent in a message.

6.1.3 Code Mobility

A location that contains a location name in its definition part can be expanded according to the *Copy* heating rule. Continuing the object example from before, an instance of $class_2$, $object_2[class_2 : \mathtt{0}]$, can be heated to include the definitions from $class_2$ and can then be heated further to also include the definitions from *class*. However, nothing can be said about when this will happen. If a `go` primitive is executed in $object_2$, the definitions from $class_2$ and *class* may or may not be included in $object_2$ when the location moves. In order to model different forms of mobility where only part of or all of a component moves, different rules are needed for how much a location definition should be expanded when it moves.

Three kinds of migration primitives are introduced: $\texttt{go}\langle b, \kappa\rangle$, $\texttt{gocore}\langle b, \kappa\rangle$, and $\texttt{gostate}\langle b, \kappa\rangle$. The migration mechanisms are the same as the original migration mechanism from the distributed join-calculus, but the reaction rules that describe their working have different side conditions. When a migration primitive is executed in a location $a[D : P]$, the only difference between the migration primitives is how many definitions will be included in a, i.e. the required form of D.

The standard `go` primitive is used to move locations including all their definitions, i.e. expanded as much as possible according to the *Copy* rule. In the above case, this would mean that $object_2$ contains all the definitions from both $class_2$ and *class* when it moves. This migration primitive is used when code components should be moved and when all of a passive or executing component should be moved (the cases in figure 3.3, 3.5, 3.6, and 3.7 in chapter 3). The following reaction rule supports this form of migration:

$$Move: \qquad a[D : P|\texttt{go}\langle b, \kappa\rangle] \vdash_{\phi} \ \| \ \vdash_{\psi b} \quad \rightarrow \quad \vdash_{\phi} \ \| \ a[D : P|\kappa\langle\rangle] \vdash_{\psi b}$$
$$(D \text{ is fully expanded})$$

The side condition means that the *Copy* rule has been applied as much as possible to a. The condition does not mean that D must not contain any location names; it only means that if D contains a location name and such a location exists, then it must be expanded according to the *Copy* rule. This is to allow a location to be defined from a location that does not exist or isn't available, and still make it possible to move the location.

The `gocore` and `gostate` primitives are used when only the state of a component should be moved (the case in figure 3.4 in chapter 3). Using one of these migration primitives to move $object_2$, the location would be moved on the form where the definition part only contains the name $class_2$.

The `gostate` primitive will only move locations of the form $a[c : P]$, i.e. locations where the definition part consists of only a single location name. This will generally be the primitive to use when modelling how Java objects are being moved. The `gocore` primitive only requires that the definition part of the moving location does not contain any unnecessary definitions, i.e. this primitive will also allow e.g. locations of the form $a[c_1 \wedge c_2 : P]$. `gostate` is simply a more restricted form of `gocore`.

In order to use one of these primitives to move a location after it has been expanded, the location needs to be contracted again. This can be done using the following *Contract* cooling rule. The rule allows the definitions in a location to be contracted to the name of a code component containing these definitions:

$$a[c \wedge D_1 : P_1] \vdash_{\phi} \ \| \ c[D_2 : \mathtt{0}] \vdash_{\psi} \leftharpoondown \ a[\,D_2 \wedge D_1 : P_1] \vdash_{\phi} \ \| \ c[D_2 : \mathtt{0}] \vdash_{\psi}$$

When a location is cooled according to this rule, it is the reversible process of heating the location according to the *Copy* rule. However, this cooling rule only allows the contraction of a set of definitions to the name of a location with a null process. No processes should be involved in the cooling because this could make unwanted contractions possible.

Two objects created from the same class definition, obj_1 and obj_2, may at one point of time have the same state. If processes were allowed to be involved in a contraction, this would allow one of the objects to be contracted to the form $obj_1[obj_2 : \mathtt{0}]$. If the second object obj_2 now changes its state, this change would be transferred to obj_1 when it again is expanded according to the *Copy* rule. This is not wanted, and the contraction is therefore only allowed to involve definitions and not processes.

When a `gocore` or `gostate` primitive is used in a location, the *Contract* rule can be applied in order to cool down the location to the required form and the rule for the migration can then be applied. When the location has arrived at the destination location, the *Copy* rule can be used to expand the location again. The reaction rules for the two migration primitives are the following:

$$\begin{array}{lccc} MoveC: & a[D : P|\mathtt{gocore}\langle b,\kappa\rangle] \vdash_{\phi} \ \| \ \vdash_{\psi b} & \rightarrow & \vdash_{\phi} \ \| \ a[D : P|\kappa\langle\rangle] \vdash_{\psi b} \\ & & (a \text{ frozen}) & \\ MoveS: & a[c : P|\mathtt{gostate}\langle b,\kappa\rangle] \vdash_{\phi} \ \| \ \vdash_{\psi b} & \rightarrow & \vdash_{\phi} \ \| \ c[D : P|\kappa\langle\rangle] \vdash_{\psi b} \end{array}$$

The side condition "a frozen" in the $MoveC$ rule means that the location should have been cooled as much as possible, particularly using the *Contract* rule. It is not required that the location only contains location names in its definition part, only that it cannot be contracted any further.

The phrase "a frozen" has for the original DRCHAM only meant that the location has been cooled as much as possible according to the `str-loc` rule. This means that all sublocations of a are in definition form instead of solution form. "a frozen" has been used as a side condition to the `str-loc` rule, and this rule will now take the following form:

$$DefSol: \quad \begin{array}{c} a[D : P] \vdash_{\phi} \ \rightleftharpoons \ \vdash_{\phi} \ \| \ \{\!|D|\!\} \vdash_{\phi a} \{\!|P|\!\} \\ (\text{no sublocation of } a \text{ is in solution form}) \end{array}$$

The side condition to the $DefSol$ rule will ensure that whenever a location is in definition form, it will also contain the definitions of all its sublocations. This will ensure that a location will bring all its sublocations when it migrates.

The migration primitives that have been introduced here, only differ in how many definitions that will be included in a location when it moves. If the moving location not is defined from any other location, the `go` and `gocore` primitives will behave the same.

6.2 Scoping

When definitions are being copied from one location to another, location definitions may also be copied. In fact, if the location, from where definitions are being copied, has any sublocations, these sublocations will be copied too. This is ensured because the location c in the *Copy* rule is in definition form (instead of solution form), and it will thus contain the definitions of all its sublocations.

The copying of a location definition will introduce multiple locations with the same name. This is not wanted because location names need to be unique in order to make message transport deterministic. When a location definition is being copied, the defined location should thus be given a new name.

The renaming of locations should be introduced in the *Copy* rule, and the scope of the new names should be determined. In order to do that, the scoping rules for the distributed join-calculus will be reviewed and the rules will then be extended to support the creation of a new scope when a location is copied.

6.2.1 Local Definitions

The scoping in the distributed join-calculus is determined through the definition of the *free variables* $fv[\cdot]$ in a process or definition, the *defined variables* $dv[\cdot]$ in a definition, and the rule for introduction of local definitions. The definition of $fv[\cdot]$ and $dv[\cdot]$ can be seen in appendix C.

Local definitions can be made with the `def` D `in` P construct. The scope of the variables defined in D is D and P; the defined variables can not be used elsewhere. The scope is determined by the rule `str-def` (which simply will be named Def in the following):

$$Def: \quad \vdash \texttt{def}\ D\ \texttt{in}\ P \ \rightleftharpoons\ D\sigma_{dv} \vdash P\sigma_{dv}$$
$$(dom(\sigma_{dv}) = dv[D],\ range(\sigma_{dv}) \text{ fresh})$$

The rule substitutes fresh distinct names for all the defined variables $dv[D]$. By "fresh" is meant a name that does not appear as a free variable in `def` D `in` P or elsewhere in the CHAM, i.e. in the context of `def` D `in` P. The substitution with fresh names will only be performed on D and P and any attempt to refer to one of the defined variables from elsewhere will thus become invalid.

The side condition "$range(\sigma_{dv})$ fresh" will also be necessary when the rule is applied in the cooling direction. The condition will then ensure that a set of definitions and processes can't be cooled down to a local definition if any of the defined variables are known elsewhere. This will also prevent a local definition from being cooled down again if any of the defined variables has been sent out of the scope.

A location definition will not introduce a new scope in the distributed join-calculus; only the `def` D `in` P construct will. This means that any port or location defined inside a location in D will be known to all of D and P, unless the definition is wrapped in a local definition construct.

With the introduction of the new language constructs, the definitions of $dv[\cdot]$ and $fv[\cdot]$ need to be extended. The final definitions can be found in section 7.2.3. The Def rule can still be used as it is.

6.2.2 Scopes Introduced by Copying

When a location definition is being copied, the copy of the defined location should be given a new name. Using the names from the *Copy* rule, this means that if c contains any sublocations, the copies of these locations should be given new names. If the definitions or processes in a contain any reference to a sublocation of c, this reference should refer to the local copy of the sublocation and not the original sublocation of c. This is motivated by the following example where a library of class definitions is defined and a JVM is given a copy of this library:

$$\begin{array}{rl} \texttt{def} & library[class_1[D_1 : \mathsf{0}] \wedge class_2[D_2 : \mathsf{0}] \wedge \ldots : \mathsf{0}] \\ \wedge & jvm[library \wedge obj[class_2 : state\langle s_0 \rangle] : \mathsf{0}] \\ \texttt{in} & \mathsf{0} \end{array}$$

When the definitions from *library* are copied to *jvm*, new names will be given to all the classes. The reference to $class_2$ in the object in the JVM, should refer to the local copy of $class_2$. The new names given to the copied classes should thus be substituted for the old names in all of *jvm*.

When a location definition is being copied, the copy should thus be given a new name. The scope of this new name should be all of the location to where the definition is being copied, including any sublocations. The introduction of new names and the scope of these names will be handled by a new version of the *Copy* rule. The rule will depend on the definition of the *free location names* $fl[\cdot]$ in a process or definition, and the *defined locations* $dl[\cdot]$ in a definition. These definitions can be found in section 7.2.3. The new *Copy* rule becomes:

$$\begin{array}{rlcl} & a[c \wedge D_1 : P_1] \vdash_\phi & \| & c[D_2 : P_2] \vdash_\psi \quad \rightharpoonup \\ Copy: & a[\,(D_2 \wedge D_1)\sigma_{dl} : (P_2|P_1)\sigma_{dl}] \vdash_\phi & \| & c[D_2 : P_2] \vdash_\psi \\ & \multicolumn{3}{c}{(dom(\sigma_{dl}) = dl[D_2],\ range(\sigma_{dl}) \text{ fresh})} \end{array}$$

The side condition "$range(\sigma_{dl})$ fresh" means that the introduced names do not appear as free location names anywhere in a or c or elsewhere in the CHAM.

The copying of code need to be done while location a is in definition form; not solution form. This is to ensure that the substitution σ_{dl} is applied to all sublocations.

If a has references to any sublocation of c, the substitution σ_{dl} will change these such that they refer to the new introduced copies and not to the original sublocations. In order to assure that no computations are carried out while the references still refer to the original locations, all substitutions, and thus all copying, should be performed before any computations take place. This is ensured by not allowing a location definition to be heated to solution form, until the location is fully expanded and all substitutions thus have been performed. The *DefSol* rule will therefore need an extra side condition:

$$DefSol: \quad \begin{array}{c} a[D : P] \vdash_{\phi} \quad \rightleftharpoons \quad \vdash_{\phi} \ \| \ \{\!|D|\!\} \vdash_{\phi a} \{\!|P|\!\} \\ \text{(no sublocation of } a \text{ is in solution form, D is fully expanded)} \end{array}$$

Structural rearrangement of the definitions in a location can only be done when the location is in solution form by using the rules `str-and` and `str-nodef` of the DRCHAM. The new side condition on the *DefSol* rule will therefore prevent such a structural rearrangement until the location is fully expanded. However, it may be necessary to rearrange the definitions in the location while it is still in definition form in order to be able to use the *Copy* rule on the location. Extra heating/cooling rules will therefore be needed.

The new rules should allow a null definition, `T`, to be introduced in a location, and they should express that "$\wedge$" is commutative and associative:

$$\begin{array}{lrcl} DefNull2: & a[D \wedge \mathtt{T} : P : \sigma] \vdash & \rightleftharpoons & a[D : P : \sigma] \vdash \\ DComm: & a[D_1 \wedge D_2 : P : \sigma] \vdash & \rightleftharpoons & a[D_2 \wedge D_1 : P : \sigma] \vdash \\ DAssoc: & a[(D_1 \wedge D_2) \wedge D_3 : P : \sigma] \vdash & \rightleftharpoons & a[D_1 \wedge (D_2 \wedge D_3) : P : \sigma] \vdash \end{array}$$

That "$\wedge$" is commutative and associative has so far been a natural part of the DRCHAM. This is because multisets are unordered and the terms in a solution therefore can be combined as wanted when the `str-and` rule is applied. This "natural" commutativity and associativity of "$\wedge$" is lost when the rearrangement of terms need to be done when a location is in definition form.

6.2.3 The Reverse Process of Name Introduction

The *Contract* rule should be modified accordingly to the *Copy* rule. The side condition "$range(\sigma_{dl})$ fresh" will also be needed on the *Contract* rule. This will ensure that a location definition can not be contracted if the location has become known outside the defining location, i.e. if a name of a copied location has been sent out of its scope. The *Contract* rule becomes:

$$Contract: \quad \begin{array}{lcl} a[c \wedge D_1 : P_1] \vdash_{\phi} & \| & c[D_2 : \mathtt{0}] \vdash_{\psi} \quad \leftharpoondown \\ a[\,(D_2 \wedge D_1)\sigma_{dl} : P_1\sigma_{dl}] \vdash_{\phi} & \| & c[D_2 : \mathtt{0}] \vdash_{\psi} \\ \multicolumn{3}{c}{(dom(\sigma_{dl}) = dl[D_2],\ range(\sigma_{dl}) \text{ fresh})} \end{array}$$

The rule doesn't say how D_1 and P_1 can be calculated from the definitions and processes in a before the contraction; the rule only says how D_1 and P_1 should be related to the original definitions and processes. Consider a CHAM with the following location definitions:

$a[D : P] \vdash_{\phi} \ \| \ c[D_2 : \mathtt{0}] \vdash_{\psi}$

If the *Contract* rule can be applied to these locations, the result will be the following:

$a[c \wedge D_1 : P_1] \vdash_\phi \ \| \ c[D_2 : \mathtt{0}] \vdash_\psi$

However, the *Contract* rule says that this is only possible if there exist a substitution σ_{dl} such that:

$$\begin{array}{lll} & (D_1 \wedge D_2)\sigma_{dl} = D & (1) \\ \wedge & P_1\sigma_{dl} = P & (2) \\ \wedge & dom(\sigma_{dl}) = dl[D_2] & (3) \\ \wedge & range(\sigma_{dl}) \text{ is fresh} & (4) \end{array}$$

If it is possible to use the *Contract* rule on the locations a and c, these conditions will be enough to uniquely determine the resulting P_1 and D_1. D_2 is known, and then $dom(\sigma_{dl})$ can be determined from (3). Condition (1) and (4) then say that D_1 should be chosen such that $D_1 \wedge D_2$ will become D if all the names belonging to $dom(\sigma_{dl})$ are substituted with fresh names (i.e. names that don't appear in D_1). This will be enough to uniquely determine D_1 and $range(\sigma_{dl})$, and then it will also be easy to determine P_1 from condition (2).

As an example consider the following location definitions:

$a[\ e[D_3 : P_3] \wedge f[D_4 : P_4] \wedge D_5 \wedge x\langle\rangle \rhd f.y\langle\rangle : e.z\langle\rangle\ ] \vdash_\phi \ \|$
$c[\ b[D_3 : P_3] \wedge d[D_4 : P_4] \wedge D_5 : \mathtt{0}\ ] \vdash_\psi$

If the *Contract* rule should be applied, we would have that $D_2 = b[D_3 : P_3] \wedge d[D_4 : P_4] \wedge D_5$ and thus $dom(\sigma_{dl}) = \{b, d\}$. Condition (1) would then give that:

$(b[D_3 : P_3] \wedge d[D_4 : P_4] \wedge D_5 \wedge D_1)\sigma_{dl} =$
$e[D_3 : P_3] \wedge f[D_4 : P_4] \wedge D_5 \wedge x\langle\rangle \rhd f.y\langle\rangle$

It can easily be determined that in order for this to hold, σ_{dl} would have to be the substitution $\{e/b, f/d\}$. By a simple matching of the expressions on the left and right hand side, we would get that D_1 is determined by:

$D_1\{e/b, f/d\} = x\langle\rangle \rhd f.y\langle\rangle$

This gives that either $D_1 = x\langle\rangle \rhd f.y\langle\rangle$ or $D_1 = x\langle\rangle \rhd d.y\langle\rangle$. The first possibility is excluded by condition (4) which says that e and f cannot appear in D_1. Now where σ_{dl} is known, condition (2) and (4) easily gives that $P_1 = b.z\langle\rangle$. The *Contract* rule can thus be applied to the locations a and c and the result will be:

$a[\ c \wedge x\langle\rangle \rhd d.y\langle\rangle : b.z\langle\rangle\ ] \vdash_\phi \ \|$
$c[\ b[D_3 : P_3] \wedge d[D_4 : P_4] \wedge D_5 : \mathtt{0}\ ] \vdash_\psi$

6.3 By-Type Bindings

As discussed in section 3.4, a Java object can be considered as having a by-type binding to its class definition. An object located at a JVM will be bound to a class definition located at that JVM. If the object moves to another JVM, it will be rebound to a similar class definition at the destination JVM.

The language constructs introduced so far allows one to model that an object is being created from a class definition and that the object moves without the class (using the `gostate` primitive). However, so far it is not possible to model that the object after the movement should be expanded using a class definition located at the destination location. A construct that allows the modelling of by-type bindings will thus be needed.

6.3.1 References Marked as By-Type

Generally, it should be possible to mark component references as being "by-type". By a component reference is meant any appearance of a location name besides the case where it is being defined, i.e. any location name appearing in the address of a message, in the content of a message, or alone in the definition part of another location. A component reference is marked as by-type with a "`" in front of the location name: $`a$.

The situation where a Java object is bound by type to its class definition could be modelled in the following way:

$$
\begin{array}{ll}
\texttt{def} & \mathit{library}[\,\mathit{class}_1[D_1 : \mathtt{0}] \wedge \mathit{class}_2[D_2 : \mathtt{0}] \wedge \ldots : \mathtt{0}\,] \\
\wedge & \mathit{jvm}_1[\,\mathit{library} \wedge \\
& \qquad \mathit{obj}[\,`\mathit{class}_2 : \mathit{state}\langle s_0\rangle \mid \texttt{gostate}\langle \mathit{jvm}_2, \kappa\rangle\,] \wedge \ldots : \ldots] \\
\wedge & \mathit{jvm}_2[\,\mathit{library} \wedge \ldots : \ldots] \\
\texttt{in} & \mathtt{0}
\end{array}
$$

The locations jvm_1 and jvm_2 each have their own copies of the class definitions. The reference to $class_2$ in obj should at first be bound to the instance at jvm_1, but when the object moves to jvm_2, the reference should change to being bound to the instance at jvm_2.

Another situation where a by-type reference would be appropriate is the case where a resource, e.g. a printer, exists at every location. An application that needs to print something would use the printer at it's own location. In the following example some general resources, including a printer, should be available at every location. An application *appl* is written to run at any location and to use the printer available at that location:

$$
\begin{array}{ll}
\texttt{def} & \mathit{resources}[\,\mathit{printer}[\mathit{print}\langle\rangle \rhd \ldots] \wedge \mathit{monitor}[\ldots] \wedge \ldots\; : \mathtt{0}\,] \\
\wedge & \mathit{appl}[\,m_1 \rhd \ldots\; `\mathit{printer.print}\langle\rangle \ldots\; : \ldots\,] \\
\wedge & \mathit{site}1[\,\mathit{resources} \wedge \mathit{appl} \wedge \ldots\; : \ldots\,] \\
\texttt{in} & \mathtt{0}
\end{array}
$$

When the resources are copied to $site_1$, every reference at $site_1$ to the *printer* and the *monitor* would be substituted with references to the local instance of that resource. If the application *appl* not yet has been "installed" at $site_1$, its reference to *printer* would not get substituted. However, by marking the *printer* reference as by-type, it will get bound to any local instance of *printer* when it is copied to $site_1$.

A component reference marked as by-type should thus always point to a local instance of that component whenever such a local instance exists. If the component with the by-type reference moves to another location, the reference should change to point to a similar component at the destination location.

A local component instance is created when a component is copied to a location. The connection from the original component name to the name of the copy will need to be saved to make it possible to determine which local instances are available at a location.

When a reference marked as by-type is actually being used (i.e. when definitions from the referenced location should be copied or when a message using that reference is about to be sent), the original component name used in the reference should be changed to the name of a local instance of that component. The reference should not be changed to the name of a local component instance until it is actually needed. This will allow a component with a by-type reference to move between locations without changing the reference.

6.3.2 The Connection from Original Components to Instances

In order to handle the connection from original component names to local instance names, a substitution will be attached to every location. The substitution should replace original component names with the names of the component instances available at the current location.

When a location is in definition form, the attached substitution will only contain the connections to the component instances *introduced* at that specific location. A location definition will henceforth have the form $a[D : P : \sigma]$. When the location is in solution form, the attached substitution will hold the connections to every component instance *available* at the location, i.e. also instances introduced in a parent location. A location in solution form will be written: $P \vdash^{\sigma}_{\phi} D$.

Initially all substitutions will be empty, but they will grow during computations as component instances are created by copying. An empty substitution is introduced into a location definition with the following heating rule:

$$DefLoc: \qquad a[D : P] \vdash \quad \rightharpoonup \quad a[D : P : \{\}] \vdash$$

When a location definition is heated to solution form, the introduced substitution will be combined with the substitutions introduced in the parent locations:

$$DefSol: \qquad \begin{array}{c} a[D : P : \sigma_2] \vdash^{\sigma_1}_{\phi} \quad \rightleftharpoons \quad \vdash^{\sigma_1}_{\phi} \;\|\; \{\!|D|\!\} \vdash^{\sigma_1\sigma_2}_{\phi a} \{\!|P|\!\} \\ \text{(no sublocation of } a \text{ is in solution form, } D \text{ is fully expanded)} \end{array}$$

The composed substitution $\sigma_1\sigma_2$ will connect original component names with the names of all the available local component instances.

The substitutions grow when components are copied. The substitution σ_{dl} applied in the *Copy* rule will contain the bindings from original components to local instances of these components. This substitution should thus be saved in the location definition.

The *Copy* rule will need to be modified further, besides the storing of the substitution σ_{dl}. By-type references should not be bound to a local component instance until it is actually needed. The substitution σ_{dl} in the *Copy* rule should thus not be applied to references marked as by-type. A special substitution function, $g_1(P, \sigma)$, is therefore introduced. The function will apply the substitution, σ, to all processes and definitions in the first argument but without substituting component references marked as by-type. The definition of $g_1(\cdot, \cdot)$ can be found in section 7.2.2.

The *Copy* rule is changed according to the above, and a first simple attempt is the following:

$$\begin{array}{ll} & a[c \wedge D_1 : P_1 : \sigma_1] \vdash_\phi \;\; \| \;\; c[D_2 : P_2 : \{\}] \vdash_\psi \qquad \rightharpoonup \\ Copy: & a[\, g_1(D_2 \wedge D_1\, ,\, \sigma_{dl}) : g_1(P_2|P_1\, ,\, \sigma_{dl}) : \sigma_1\sigma_{dl}] \vdash_\phi \;\; \| \;\; c[D_2 : P_2 : \{\}] \vdash_\psi \\ & (dom(\sigma_{dl}) = dl[D_2],\, range(\sigma_{dl}) \text{ fresh}) \end{array}$$

The rule is presented with an empty substitution in the component c. The case where the copied component has a non-empty substitution needs a little further consideration. Consider the following example:

$$\begin{array}{ll} \texttt{def} & resources[\, printer[print\langle\rangle \rhd \ldots] \wedge monitor[\ldots] : \mathsf{0}\,] \\ \;\wedge & computer[\, resources \wedge jvm[\ldots] : \mathsf{0}\,] \\ \;\wedge & comp_1[\, computer : printer.print\langle\rangle\,] \\ \texttt{in} & \mathsf{0} \end{array}$$

When the definitions from *resources* are copied to *computer*, new names will be introduced for *printer* and *monitor*. The *computer* location will thus get an attached substitution of the form

$$\sigma_2 = \{printer_1/printer, monitor_1/monitor\}$$

When the definitions from *computer* then are copied to the location $comp_1$, the introduced substitution, σ_{dl}, will be of the form

$$\sigma_{dl} = \{printer_2/printer_1, monitor_2/monitor_1, jvm_2/jvm\}$$

When this substitution is applied to the other processes in $comp_1$, the reference to *printer* will not be changed; the substitution σ_2 will need to be applied first. The *Copy* rule will therefore get the following form:

$$\begin{array}{ll} & a[c \wedge D_1 : P_1 : \sigma_1] \vdash_\phi \;\; \| \;\; c[D_2 : P_2 : \sigma_2] \vdash_\psi \qquad \rightharpoonup \\ Copy: & a[\, g_1(D_2 \wedge D_1\, ,\, \sigma_2\sigma_{dl}) : g_1(P_2|P_1\, ,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \vdash_\phi \;\; \| \;\; c[D_2 : P_2 : \sigma_2] \vdash_\psi \\ & (dom(\sigma_{dl}) = dl[D_2],\, range(\sigma_{dl}) \text{ fresh}) \end{array}$$

The substitution σ_2 is applied to D_1 and P_1 to ensure that references to subcomponents of c really refers to these components and not the original names of these subcomponents. However, such references will in all cases be renamed again by σ_{dl}.

6.3.3 Substitution of By-type References

When a component reference marked as by-type is about to be used, the reference should be substituted with the name of a local component instance.

By-type references in pending messages should be substituted at the moment where the message actually is being sent. This happens when the message appears as result of a reduction. As long as the message only appears in the guarded process of a reduction definition, all by-type references should keep their original name. At the moment where the reduction is performed, the by-type references should be replaced.

The replacing of by-type references is handled by the substitution function $g_2(P,\sigma)$. g_2 will use the substitution σ to replace all by-type references in P with names of local component instances. However, the substitution will not be applied to definitions within P because a reference in a definition not actually is being used yet. The definition of $g_2(\cdot,\cdot)$ can be found in section 7.2.2.

The replacement of by-type references in messages should happen when a reduction is performed, and the *Red* rule should thus be changed:

$$\mathit{Red}: \quad \begin{array}{c} J \rhd P \vdash^{\sigma} J\sigma_{rv} \;\rightarrow\; J \rhd P \vdash^{\sigma} g_2(P\sigma_{rv},\sigma) \\ (\mathit{dom}(\sigma_{rv}) = rv[J]) \end{array}$$

First, the substitution from the received variables $rv[J]$ to the actually transmitted names, σ_{rv}, is applied to the process P. Next, all by-type references are replaced according to the substitution σ attached to the current location.

Except for messages, a by-type reference can also appear in a definition where a location name is used to copy definitions. Such a by-type reference should not be resolved to a local component instance until the copying actually is being performed. The *Copy* rule should thus be changed to also cover the case where the location reference is marked as by-type. The symbols $\alpha, \beta, \ldots$ will in the following be used to denote both marked and unmarked location names. The *Copy* rule becomes:

$$\mathit{Copy}: \quad \begin{array}{c} a[\beta \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_3}_{\phi} \;\; \| \;\; c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \qquad \rightharpoonup \\ a[\, g_1(D_2 \wedge D_1 \,,\, \sigma_2\sigma_{dl}) : g_1(P_2|P_1 \,,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \vdash^{\sigma_3}_{\phi} \;\; \| \;\; c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \\ (g_2(\beta,\sigma_3\sigma_1) = c,\; \beta \neq \grave{}c,\; \mathit{dom}(\sigma_{dl}) = dl[D_2],\; \mathit{range}(\sigma_{dl}) \text{ fresh}) \end{array}$$

The new side condition $g_2(\beta,\sigma_3\sigma_1) = c$ says that the location c should be a local instance of the location referenced by β. If β isn't marked as by-type, it will be β itself from where definitions will be copied, $g_2(\beta,\sigma_3\sigma_1) = \beta$. If the reference β is marked as by-type but no local instance exists, then it should not be possible to perform the copying. That is the reason for the second side condition $\beta \neq \grave{}c$.

The *Contract* rule will also need to be modified accordingly:

$$\mathit{Contract}: \quad \begin{array}{c} a[\grave{}b \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_3}_{\phi} \;\; \| \;\; c[D_2 : 0 : \sigma_2] \vdash^{\sigma_4}_{\psi} \qquad \leftharpoondown \\ a[\, g_1(D_2 \wedge D_1 \,,\, \sigma_2\sigma_{dl}) : g_1(P_1 \,,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \vdash^{\sigma_3}_{\phi} \;\; \| \;\; c[D_2 : 0 : \sigma_2] \vdash^{\sigma_4}_{\psi} \\ (g_2(\grave{}b,\sigma_3\sigma_1) = c,\; b \notin \mathit{range}(\sigma_3\sigma_1),\; b \notin \mathit{range}(\sigma_4), \\ \mathit{dom}(\sigma_{dl}) = dl[D_2],\; \mathit{range}(\sigma_{dl}) \text{ fresh},\; \mathit{range}(\sigma_2) \text{ fresh in } a) \end{array}$$

The side conditions, $g_2(`b, \sigma_3\sigma_1) = c$, $b \notin range(\sigma_3\sigma_1)$, and $b \notin range(\sigma_4)$, are necessary in order to determine what $b$ should be in case there are several component instances with the definition $D_2$. The condition $g_2(`b, \sigma_3\sigma_1) = c$ says that c is either a local instance of the component with the name b, or c equals b. The condition $b \notin range(\sigma_3\sigma_1)$ ensures that b isn't the name of a component instance in a. The condition $b \notin range(\sigma_4)$ ensures that if $b = c$ then b isn't the name of a componant instance at some other location. Thus b have to be the name of the original component.

When a set of definitions, D_2, are contracted to a location name, b, the location name will always be marked as by-type. This will be the case even though the definitions originate from an expansion from an unmarked location name. However, the marking of the location name, $`b$, can not hurt because the location $a$ always can be expanded again to its original form. Only if the location is being moved using the `gostate` primitive, will the by-type marking make a difference, and there is no reason to use the `gostate` primitive unless the reference to the code component is marked as by-type.

The *Contract* rule still doesn't say anything about how D_1 and P_1 should be computed, as discussed in section 6.2.3. There are now two unknown substitutions, σ_{dl} and σ_1 but it will still be possible to uniquely determine D_1 and P_1. This is possible because of the extra side condition "$range(\sigma_2)$ fresh in a" and because the rule gives the extra condition that $\sigma_1\sigma_2\sigma_{dl}$ should equal the original substitution in location a. The condition "$range(\sigma_2)$ fresh in a" means that $range(\sigma_2) \cap (fl[D_1] \cup fl[P_1]) = \emptyset$.

The cases handled by the above rules, are the only cases where by-type references should appear. If a by-type reference is used anywhere else, then it will not become replaced with a component name and it will thus not make any sense. A location should therefore not be defined with a by-type reference in its process part. A by-type reference should only appear within a definition.

6.3.4 Component Movement

When a component moves, its by-type references will stay the same; just like every other references. It is the component's parent location, and thus also the attached substitution from original components to component instances, that will change. This will result in by-type references being bound to other components when they are used at the destination location. The *Move* rules can thus be used as they are.

However, not all components should be allowed to move any more. A component instance can not leave its parent location without "messing up" things. If the parent location knows the component as a local instance of another component, then it will still bind by-type references to the component, even after it has moved to another location. A by-type reference should always be bound to a local component instance and the component instance should thus not be allowed to move if there are any bindings to it.

To avoid the movement of a component instance, two extra side conditions will need to be added to the *Move* rule:

$$Move: \begin{array}{l} a[D : P|\mathtt{go}\langle b,\kappa\rangle : \sigma_1] \vdash^{\sigma_2}_{\phi} \;\|\; \vdash_{\psi b} \;\;\rightarrow\;\; \vdash^{\sigma_2}_{\phi} \;\|\; a[D : P|\kappa\langle\rangle : \sigma_1] \vdash_{\psi b} \\ (D \text{ is fully expanded, } a \notin range(\sigma_2), \text{ for all } c \in dl[D]\text{: } c \notin range(\sigma_2)) \end{array}$$

The extra side conditions ($a \notin range(\sigma_2)$, for all $c \in dl[D]$: $c \notin range(\sigma_2)$) say that a location can not move if itself or a sublocation is known to the parent location as a local component instance. These side conditions will also be needed on the rules for the other migration primitives.

How the movement of a component instance can be made possible, despite the extra side conditions, will be considered in the next section.

6.4 Component Instances

So far component instances have been created simply when location definitions have been copied. In general it would be useful to be able to model that a certain location should count as an instance of another location even though it isn't created as a copy of that location. Therefore a new primitive will be introduced that can be used to mark a location as being an instance of another location

6.4.1 Locations Marked as Component Instances

The *Copy* rule is the rule that can be used to create component instances. When all the definitions from one location is copied to another location, component instances will be created for all the sublocations. However, it is only the sublocations that will be connected with a component instance through the substitution σ_{dl}; the location from where the definitions are being copied will not become connected with the other location. Consider the following example:

$$\begin{array}{rl} \mathtt{def} & printercode[printer[D : P] : 0] \\ \wedge & site[printercode \wedge \ldots : \ldots] \\ \mathtt{in} & 0 \end{array}$$

This will create an instance of the *printer* component at the *site* location because the *printer* location is a sublocation of the *printercode* location from where definitions are being copied. The following will, however, not create a *printer* instance:

$$\begin{array}{rl} \mathtt{def} & printer[D : P] \\ \wedge & site[colorprinter[printer \wedge \ldots] \wedge \ldots : \ldots] \\ \mathtt{in} & 0 \end{array}$$

In this situation it would be useful if the *colorprinter* component could count as an instance of the general *printer* component. In general it would be useful if one could copy the definitions from one component to another component, maybe add some extra definitions, and make the latter component count as an instance of the former component.

To make it possible to specify that a component should count as an instance of another component, the $\mathtt{subst}\langle\cdot\rangle$ primitive is added to the language. If a $\mathtt{subst}\langle\cdot\rangle$ appears in the process part of a location, the parent location is told that the location should count as an instance of the component b. Using the `subst` primitive, it is possible to specify that the *colorprinter* in the above example should count as an instance of the *printer* component:

$$\begin{array}{rl} \mathtt{def} & printer[D : P] \\ \wedge & site[colorprinter[printer \wedge \ldots : \mathtt{subst}\langle printer\rangle] \wedge \ldots : \ldots] \\ \mathtt{in} & \mathtt{0} \end{array}$$

This should make *colorprinter* known as an instance of *printer* in all of the *site* location.

The `subst` primitive will also be useful if a component instance is created at one location but should be moved to another location. The dynamic class downloading in Java requires that a copy of a class definition can be moved from an http server to a JVM and be made to count as an instance of the class at the JVM. This is possible with the `subst` primitive. The following shows how the http server can be modelled:

$$\begin{array}{l} http[\ c_1[D_1 : \mathtt{0}] \wedge c_2[D_2 : \mathtt{0}] \wedge c_3[D_3 : \mathtt{0}] \wedge \ldots \wedge \\ \qquad fetch\langle class, jvm\rangle \triangleright\ \ \mathtt{def}\ \ newc[\ class : \mathtt{subst}\langle class\rangle \mid \mathtt{go}(jvm); \mathtt{0}\ ] \\ \qquad\qquad\qquad\qquad\qquad\ \ \mathtt{in}\ \ \mathtt{0} \\ \quad : \mathtt{0}\ ] \end{array}$$

When the http server receives a request for a specific class, the server creates a copy of the class and marks the copy with a $\mathtt{subst}\langle class\rangle$. The `subst` primitive will tell any parent location of *newc* that *newc* is an instance of the requested class.

The `subst` primitive makes it possible to move a component that is an instance of another component. This is possible because the connection from the original component to the component instance is saved within the component instance and not in the parent location's attached substitution. In general, it should be possible to move any component instance by changing the connection in the parent location's substitution into a `subst` primitive within the component instance. The component instance can then move, and when it arrives at the destination location, the `subst` primitive can be changed back to a substitution in the new parent location.

6.4.2 The *Subst* rule

The chemical semantics of the language should be extended with a rule that reflects the intended meaning of the `subst` primitive. When a $\mathtt{subst}\langle b\rangle$ appears inside the definition of a location c, it should have the same effect as if location c had just been created as a copy of location definition b. This means that the substitution $\{c/b\}$ should be applied using the substitution function $g_1(\cdot,\cdot)$ everywhere inside the parent location of c.

If location c has introduced new names for any subcomponents, these subcomponents should also become known as component instances in the parent location. This is to make this way of introducing component instances similar to when they are introduced by pure copying; when components are copied, all their subcomponents will also become known as component instances in the location to where they are copied.

This is expressed in the following heating rule:

$$
\begin{array}{ll}
 & a[\ c[\ D_2 : P_2|\texttt{subst}\langle b\rangle : \sigma_2\ ]\ \wedge D_1 : P_1 : \sigma_1\ ]\quad \vdash^{\sigma_3} \qquad \rightharpoonup \\
\mathit{Subst}: & a[\ c[\ g_1(D_2, \{c/b\}) : g_1(P_2, \{c/b\}) : \sigma_2\ ] \wedge \\
 & \quad g_1(D_1, \{c/b\}\sigma_2) : g_1(P_1, \{c/b\}\sigma_2) : \sigma_1\{c/b\}\sigma_2\ ]\quad \vdash^{\sigma_3}
\end{array}
$$

The parent location a need to be in definition form when the `subst` primitive is changed into a substitution. This is to ensure that the substitutions are applied to all sublocations of a.

The substitution $\{c/b\}$ is also applied inside location c in case the location has any subcomponents that use the original component name b. This is the situation in the following example where a printer makes use of a subcomponent for testing purposes:

$$
\begin{array}{ll}
\texttt{def} & \mathit{printer}[\ \mathit{print}\langle x\rangle \triangleright \ldots \wedge \\
 & \qquad \mathit{subcomp}[\ \mathit{test}\langle\rangle \triangleright \mathit{printer.print}\langle \mathit{testpage}\rangle \ldots\ ]\ \wedge \ldots\ ] \\
\wedge & \mathit{site}[\ \mathit{colorprinter}[\ \mathit{printer} \wedge \ldots : \texttt{subst}\langle \mathit{printer}\rangle\ ]\ \wedge \ldots : \ldots\ ] \\
\texttt{in} & \texttt{0}
\end{array}
$$

When the definitions from *printer* have been copied to *colorprinter* and the *colorprinter* location is heated according to the *Subst* rule, the substitution $\{\mathit{colorprinter}/\mathit{printer}\}$ will also be applied to the copy of the sublocation *subcomp*. The subcomponent of the *colorprinter* will thus communicate with its parent location *colorprinter* and not the original parent location *printer*.

The above example also shows that a side condition "D_2 fully expanded" will be necessary on the *Subst* rule. The definitions from *printer* should be copied to *colorprinter* before the *Subst* rule is applied; otherwise the printer reference used for the copying would also be changed and the copying can not be done.

The *Subst* rule should be made reversible such that a component instance, which is created using a `subst` primitive, can be cooled down with the *subst* rule and be made to fulfil the side conditions of the *Move* rules. This will allow a component instance to remove all bindings to itself and its subcomponents at its current location, then move to another location, and finally make itself and its subcomponents become known again at the destination location.

Certain conditions should be fulfilled before a location can be cooled using a reversible *Subst* rule. The sublocations of c and c itself should not be known as component instances in the parent location a after the rule has been applied. The rule should thus only be used in the cooling direction when it is possible to fulfil the condition $(\{c\} \cup dl[D_2]) \cap range(\sigma_3\sigma_1) = \emptyset$. This will not always be the case, and except for the case where $dl[D_2] = \emptyset$ will it only be possible when c has been introduced as a component instance using a `subst` primitive.

The final *Subst* rule becomes:

$$\begin{array}{ll} & a[\ c[\ D_2 : P_2|\mathtt{subst}\langle b\rangle : \sigma_2\ ]\ \wedge D_1 : P_1 : \sigma_1\ ] \quad \vdash^{\sigma_3} \qquad \rightleftharpoons \\ \textit{Subst}: & a[\ c[\ g_1(D_2, \{c/b\}) : g_1(P_2, \{c/b\}) : \sigma_2\ ]\ \wedge \\ & \quad g_1(D_1, \{c/b\}\sigma_2) : g_1(P_1, \{c/b\}\sigma_2) : \sigma_1\{c/b\}\sigma_2\ ] \quad \vdash^{\sigma_3} \\ & \quad (D_2 \text{ is fully expanded, } (\{c\} \cup dl[D_2]) \cap range(\sigma_3\sigma_1) = \emptyset) \end{array}$$

Similarly to the *Contract* rule, the *Subst* rule does not describe how D_1, P_1, D_2, P_2, and b can be computed from the original location definitions when the rule is applied in the cooling direction. Consider a CHAM with the following solution:

$$a[\ c[\ D : P : \sigma_2\ ]\ \wedge D_3 : P_3 : \sigma\ ] \quad \vdash^{\sigma_3}$$

The *Subst* rule says that the solution can be cooled to a solution:

$$a[\ c[\ D_2 : P_2|\mathtt{subst}\langle b\rangle : \sigma_2\ ]\ \wedge D_1 : P_1 : \sigma_1\ ] \quad \vdash^{\sigma_3}$$

but only if it is possible to find a D_1, a P_1, a σ_1, a D_2, a P_2, and a b such that the following conditions are fulfilled:

$$\begin{array}{lll} & g_1(D_2, \{c/b\}) = D & (1) \\ \wedge & g_1(P_2, \{c/b\}) = P & (2) \\ \wedge & g_1(D_1, \{c/b\}\sigma_2) = D_3 & (3) \\ \wedge & g_1(P_1, \{c/b\}\sigma_2) = P_3 & (4) \\ \wedge & \sigma = \sigma_1\{c/b\}\sigma_2 & (5) \end{array}$$

Condition (5) is the condition that determines b and σ_1. However, it is only possible to determine b if σ only contains one pair where c is substituted for some other name. Condition (1), (2), (3), and (4) will be used to determine respectively D_2, P_2, D_1 and P_1. An extra condition $(\{c\} \cup range(\sigma_2)) \cap (fl[D_1] \cup fl[P_1]) = \emptyset$ will be necessary in order to uniquely determine D_1 and P_1.

6.4.3 Modification of Other Rules

It should not be possible to heat a location definition to solution form until all necessary substitutions have been applied in the location. That was the reason for giving the *DefSol* rule the side condition "D is fully expanded", which ensures that all substitutions that can be introduced using the *Copy* rule will have been applied before the *DefSol* rule can be used.

It should be similarly for all substitutions that can be introduced using the *Subst* rule. It should not be possible to heat a location definition to solution form until all `subst` primitives have been changed to substitutions. The *DefSol* rule will therefore need an extra side condition that can guarantee that a location isn't heated to solution form if it contains a sublocation showing a `subst` primitive. The cooling in the opposite direction should however be allowed because a `subst` primitive should be allowed to appear during computations. However, in such a case it can not be guarantied that the substitution will take effect before further computations are carried out.

The *DefSol* rule will need to be split in two rules; a cooling rule and a heating rule:

$$DefSol: \quad a[D : P : \sigma_2] \vdash^{\sigma_1}_{\phi} \;\rightharpoonup\; \vdash^{\sigma_1}_{\phi} \;\|\; \{\!|D|\!\} \vdash^{\sigma_1\sigma_2}_{\phi a} \{\!|P|\!\}$$
(D is fully expanded, no sublocation of a is on the form $b[D_1 : P_1 | subst\langle c\rangle]$)

$$SolDef: \quad a[D : P : \sigma_2] \vdash^{\sigma_1}_{\phi} \;\leftharpoondown\; \vdash^{\sigma_1}_{\phi} \;\|\; \{\!|D|\!\} \vdash^{\sigma_1\sigma_2}_{\phi a} \{\!|P|\!\}$$
(no sublocation of a is in solution form)

In this way, all substitutions will have been applied before a location can be heated to solution form and any computation take place.

Because of the need for all substitutions to take place before a location is heated to solution form, further rules will need to be modified. The current version of the *Move* rule will move a location from a location in solution form to another location in solution form. The rule should be changed such that a location moves from a location in definition form to another location in definition form. This will be necessary in order to allow the movement of a location containing a `subst` primitive and to make sure that the corresponding substitution is applied at the destination location before further computations are carried out. The *Move* rule is changed to the following:

$$\begin{array}{ll} & a_1[\, c[D_2 : P_2 | \mathtt{go}\langle a_2, \kappa\rangle : \sigma_2] \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_{\phi} \;\|\; a_2[D_3 : P_3 : \sigma_3] \vdash_{\psi} \;\longrightarrow \\ Move: & a_1[D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_{\phi} \;\|\; a_2[\, c[\, D_2 : P_2 | \kappa\langle\rangle : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash_{\psi} \\ & (D_2 \text{ is fully expanded, } c \notin range(\sigma_4\sigma_1), \text{ for all } b \in dl[D_2]\text{: } b \notin range(\sigma_4\sigma_1)) \end{array}$$

The rules for the other migration primitives should be modified accordingly.

Because a location now moves from within one location definition to within another, a `go` primitive can no longer be used in one of the top-level locations; there need to be a surrounding location definition. A system should thus always be modelled with stationary top-level locations, e.g. domains or other stationary computing environments.

The new side condition on the *DefSol* rule means that a location with a sublocation showing a `subst` primitive ($a[\, c[\ldots : \mathtt{subst}\langle b\rangle] \wedge \ldots]$) cannot be heated to solution form until the corresponding substitution has been applied. One of the side conditions on the *Subst* rule says that this can not happen until the sublocation (c) is fully expanded. However, the *Copy* rule can only be used on a location in solution form and can thus not be applied. In order to anyway allow such a location to be heated, an extra version of the *Copy* rule will be added. The new version allows definitions to be copied to a location appearing inside another location definition:

$$\begin{array}{ll} & d[\, a[\beta \wedge D_1 : P_1 : \sigma_1] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi} \;\|\; c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \;\rightharpoonup \\ Copy2: & d[\, a[\, g_1(D_2 \wedge D_1 \,,\, \sigma_2\sigma_{dl}) : g_1(P_2|P_1 \,,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi} \\ & \qquad \| \;\; c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \\ & (g_2(\beta, \sigma_4\sigma_3\sigma_1) = c,\ \beta \neq \grave{}c,\ dom(\sigma_{dl}) = dl[D_2],\ range(\sigma_{dl}) \text{ fresh}) \end{array}$$

This rule does not cover the case where a and c are sibling locations, and a third *Copy* rule will therefore also be needed:

$$\begin{array}{ll} & d[\, a[\beta \wedge D_1 : P_1 : \sigma_1] \wedge c[D_2 : P_2 : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi} \;\rightharpoonup \\ Copy3: & d[\, a[\, g_1(D_2 \wedge D_1 \,,\, \sigma_2\sigma_{dl}) : g_1(P_2|P_1 \,,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \wedge \\ & \quad c[D_2 : P_2 : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi} \\ & (g_2(\beta, \sigma_4\sigma_3\sigma_1) = c,\ \beta \neq \grave{}c,\ dom(\sigma_{dl}) = dl[D_2],\ range(\sigma_{dl}) \text{ fresh}) \end{array}$$

A new version of the *DefLoc* rule will also be needed because it can be necessary to introduce an empty substitution in a location definition appearing inside another location definition:

$$DefLoc2: \qquad b[\ a[D : P] \wedge D_2 : P_2 : \sigma] \vdash \quad \rightharpoonup \quad b[\ a[D : P : \{\}] \wedge D_2 : P_2 : \sigma] \vdash$$

This rule would need to be used before any of the above *Copy* rules can be used.

Finally, it will also be necessary to introduce rules that allow structural rearrangement of the definitions in a location appearing inside another location definition. This can be necessary in order to be able to apply one of the above copy rules to a location. The following rules should therefore be added:

$$\begin{array}{ll}
DefNull3: & b[\ a[D \wedge \mathtt{T} : P : \sigma] \ \wedge D_2 : P_2 : \sigma_2\ ] \vdash \quad \rightleftharpoons \\
 & b[\ a[D : P : \sigma] \ \wedge D_2 : P_2 : \sigma_2\ ] \vdash \\
DComm2: & b[\ a[D_1 \wedge D_2 : P : \sigma] \ \wedge D_3 : P_3 : \sigma_3\ ] \vdash \quad \rightleftharpoons \\
 & b[\ a[D_2 \wedge D_1 : P : \sigma] \ \wedge D_3 : P_3 : \sigma_3\ ] \vdash \\
DAssoc2: & b[\ a[(D_1 \wedge D_2) \wedge D_3 : P : \sigma] \ \wedge D_4 : P_4 : \sigma_4\ ] \vdash \quad \rightleftharpoons \\
 & b[\ a[D_1 \wedge (D_2 \wedge D_3) : P : \sigma] \ \wedge D_4 : P_4 : \sigma_4\ ] \vdash
\end{array}$$

Chapter 7

The Mobility Language

The language that was constructed in chapter 6 will here be presented in its entirety. The language, henceforth called the Mobility Language, will also include the failure handling part from the distributed join-calculus, which was left out in the last chapter.

The syntax of the Mobility Language will be presented in section 7.1. The semantics of the language is described in terms of a chemical abstract machine, the MLCHAM. The chemical rules of the MLCHAM make use of substitutions, auxiliary functions, etc. These will be defined formally in section 7.2 before the MLCHAM is presented in section 7.3.

7.1 Syntax

The Mobility Language is a modification of the distributed join-calculus and as such has the same basic constructs. The Mobility Language adds syntax that allows copying of definitions from one location to another, creation of component instances, and use of by-type bindings. The language further operates with three kinds of migration primitives instead of just one, allowing a location to migrate bringing none, part of or all of its reduction definitions.

The existence of the following set of names will be assumed:

- $\mathcal{N}$, an infinite set of port names ranged over by the variables $x, y, \ldots$ A port has an arity telling the number of values contained in a message to that port.
- $\mathcal{L}$, an infinite set of location names ranged over by the variables $a, b, \ldots$ A marker $\Omega \notin \mathcal{L}$ is used to tag failed locations. For every $a \in \mathcal{L}$, ϵa denotes either a or Ωa. Finite strings of such possibly tagged location names are denoted by $\phi, \psi, \ldots$

Port names, location names, and combinations where a port name is prefixed with a location name, $a.x$, constitute the *values* of the Mobility Language.

All values appearing in a process in other places than where the port names and location names are defined, are considered *references*. A reference to a location can be marked as by-type: $`a$. The variables $\alpha, \beta, \ldots$ will be used to range over both marked and unmarked location references, i.e. $\alpha$ can mean either $a$ or $`a$. κ will be used to denote a reference on the form x, $a.x$ or $`a.x$, i.e. a reference that can be used as the addressing part of a message. $u, v, \ldots$ denotes any kind of reference, and $\tilde{v}$ denotes a tuple of references.

The terms of the Mobility Language are processes P, definitions D, and join patterns J. They are defined by the following abstract syntax:

$P ::=$	$P \vert P$	composition	$D ::=$	$D \wedge D$	co-definition
$\vert$	$\texttt{def}\ D\ \texttt{in}\ P$	definition	$\vert$	$J \triangleright P$	local rule
$\vert$	$\kappa\langle\tilde{v}\rangle$	message	$\vert$	$\texttt{T}$	inert definition
$\vert$	$\texttt{0}$	inert process	$\vert$	$a[D : P]$	initial sub-location
$\vert$	$\texttt{go}\langle\alpha, \kappa\rangle$	migration	$\vert$	$a[D : P : \sigma]$	sub-location
$\vert$	$\texttt{gocore}\langle\alpha, \kappa\rangle$	core migration	$\vert$	$\Omega a[D : P : \sigma]$	dead sub-location
$\vert$	$\texttt{gostate}\langle\alpha, \kappa\rangle$	state migration	$\vert$	α	location copy
$\vert$	$\texttt{subst}\langle a\rangle$	local instance			
$\vert$	$\texttt{halt}\langle\rangle$	termination	$J ::=$	$J \vert J$	join-pattern
$\vert$	$\texttt{fail}\langle\alpha, \kappa\rangle$	failure detection	$\vert$	$x\langle\tilde{v}\rangle$	message pattern

The process $P|P$ denotes parallel composition of processes, $\texttt{0}$ is the inert process, and $\texttt{def}\, D\, \texttt{in}\, P$ is a local definition where the port names and location names defined in D only are known in D and in P.

A definition can consist of several definitions, $D \wedge D$, it can be the inert definition, $\texttt{T}$, it can be a reduction definition, it can be a location definition, and it can be a location reference.

A reduction definition $J \triangleright P$ defines certain port names. The reduction describes how messages to these ports can start a new process P. A reduction definition is similar to a function, and a message to the ports defined in J corresponds to the invocation of the function.

A location definition $a[D : P]$ defines a location with the name a, the definitions D, and the processes P. A location can represent a component or a location. Reductions can take place in a location, and messages can be sent to other locations. A location definition can also have the form $a[D : P : \sigma]$ where σ is a substitution from location names to local instances of these locations. A location definition of the form $\Omega a[D : P : \sigma]$ is a dead location that no message or location can leave.

A definition α in a location means that all definitions from the location denoted by α should be copied to this location. If the copied definitions define any sub-locations, these new locations will count as local instances of the original locations.

The process $\kappa\langle\tilde{v}\rangle$ is a message to a port. A message can be addressed to a port in the current location or it can be addressed to a port in another location. The port name must in the latter case be prefixed with the name of the destination location whereas the port name is enough in the first case. By-type references in a message will, at the time where the message is sent, be substituted with the name of a local instance of the referenced location.

The processes $\texttt{go}\langle\alpha,\kappa\rangle$, $\texttt{gocore}\langle\alpha,\kappa\rangle$, and $\texttt{gostate}\langle\alpha,\kappa\rangle$ are migration primitives that will make the enclosing location move to the location denoted by α. Upon arrival to the destination location, a message will be sent to the port denoted by κ. When a location is moved using the `gostate` primitive, its definition part will consist of a single location reference. When a location is moved using the `gocore` primitive, as many of its definitions as possible will have been changed to location references. When a location is moved using the `go` primitive, all copying of definitions from other locations will have been performed.

The process $\texttt{subst}\langle a\rangle$ makes the enclosing location count as an instance of the location a. The process $\texttt{halt}\langle\rangle$ will make the enclosing location fail, whereas the process $\texttt{fail}\langle\alpha,\kappa\rangle$ can detect if the location denoted by α fails. A message will in that case be sent to the port denoted by κ.

Besides the syntax given by the above abstract syntax, the Mobility Language also has syntactic sugar that supports sequencing of "function calls" and return values. The syntactic sugar simply hides the use of continuation channels.

A reduction definition of a port that takes a continuation channel as the last argument and somewhere in the guarded process sends a message to this port, $x\langle\tilde{v},\kappa\rangle \triangleright \ldots \kappa\langle\tilde{u}\rangle \ldots$, can with the syntactic sugar be written as:

$$x(\tilde{v}) \triangleright \ldots \texttt{reply}\ \tilde{u}\ \texttt{to}\ x \ldots$$

A local definition of a port and the sending of a message where this port is used as a continuation, $a[\ldots \texttt{def}\, x\langle\rangle \triangleright P\ \texttt{in}\ \kappa\langle\tilde{v}, a.x\rangle \ldots]$, will be allowed to be written as:

$$a[\ldots \kappa(\tilde{v}); P \ldots]$$

The same way of writing will be allowed for the local definition of a continuation channel used in the migration primitives or the `fail` primitive, e.g. $\texttt{go}(a); P$. In the case where a tuple of values is returned on the continuation channel, $a[\ldots \texttt{def}\, x\langle\tilde{u}\rangle \triangleright P\ \texttt{in}\ \kappa\langle\tilde{v}, a.x\rangle \ldots]$, it can be written as:

$$a[\ldots \texttt{let}\ \tilde{u} = \kappa(\tilde{v})\ \texttt{in}\ P \ldots]$$

Besides having a structure given by the above abstract syntax, there are certain conditions that a well-formed process must satisfy:

- All messages should respect the arity of the defined ports.
- Location names can only be defined once in a scope (whereas the same port name can be defined in several locations in the same scope).
- By-type references are only allowed in guarded processes and in definitions where they denote location copying (i.e. not in the top-level process, not in the process part of a location, and not in join-patterns).

- The names used for the received variables in a join-pattern (the received variables are the names in $rv[J]$ as defined in section 7.2.3) must not be used as location names elsewhere (because the substitution of a by-type reference in the guarded process could interfere with the names of the received variables).
- A name must not appear twice as a received variable in the same join-pattern (i.e. a join-pattern cannot be used to match the content of messages, e.g. the join pattern $x\langle v\rangle \mid y\langle v\rangle$ is not allowed.)
- A migration primitive or a `subst` primitive should only appear in a location that has a parent location.

7.2 Auxiliary Definitions

The following sections will formally define substitution, the substitution functions $g_1(\cdot,\sigma)$ and $g_2(\cdot,\sigma)$, and the sets of free, defined, and received names, $fv[\cdot]$, $fl[\cdot]$, $dv[\cdot]$, $dl[\cdot]$, and $rv[\cdot]$.

7.2.1 Substitution

A substitution is a mapping from names to names. It will here be defined following the definition in [NM00].

Syntactically a substitution is a set of pairs of names:

$$\sigma = \{b_1/a_1, b_2/a_2, \ldots, b_n/a_n\}$$

where $\forall i, j \leq n : i \neq j \Rightarrow a_1 \neq a_j$. The domain, $dom(\cdot)$, of a substitution $\{b_1/a_1, b_2/a_2, \ldots, b_n/a_n\}$ is the set $\{a_1, a_2, \ldots, a_n\}$ and the range, $range(\cdot)$, of the substitution is the set $\{b_1, b_2, \ldots, b_n\}$.

The application $a\sigma$ of a substitution σ to a name a is defined as:

$$a\sigma = \begin{cases} b \text{ if } b/a \in \sigma \\ a \text{ if } a \notin dom(\sigma) \end{cases}$$

When a substitution σ is applied to a process, a definition, a join pattern or a reference, every occurrence of the names in $dom(\sigma)$ will be replaced. The application of a substitution to a process, a definition, a join pattern or a reference is formally defined in table 7.1.

Two substitutions, σ_1, σ_2, can be composed to a single substitution $\sigma_1\sigma_2$. The effect of applying a composed substitution should be the same as the effect of first applying the first substitution and then the second substitution:

$$a(\sigma_1\sigma_2) = (a\sigma_1)\sigma_2$$

The composition $\sigma_1\sigma_2$ of the substitutions

$$\sigma_1 = \{b_1/a_1, b_2/a_2, \ldots, b_n/a_n\} \quad \text{and}$$
$$\sigma_2 = \{d_1/c_1, d_2/c_2, \ldots, d_m/c_m\}$$

is the substitution:

$$\sigma_1\sigma_2 = \{(b_1\sigma_2)/a_1, (b_2\sigma_2)/a_2, \ldots, (b_n\sigma_2)/a_n, d_1/c_1, d_2/c_2, \ldots, d_m/c_m\}$$

The two substitutions should have disjoint domains, $dom(\sigma_1) \cap dom(\sigma_2) = \emptyset$; otherwise the pairs d_i/c_i for which $c_i \in dom(\sigma_1)$ should be removed from the composed substitution.

Two substitutions are equal if they contain the same pairs of names.

$v:$	$(\grave{}a)\sigma$	$\stackrel{\text{def}}{=} \grave{}(a\sigma)$	$(\alpha.x)\sigma$	$\stackrel{\text{def}}{=} (\alpha\sigma).(x\sigma)$	
$J:$	$(J_1 \vert J_2)\sigma$	$\stackrel{\text{def}}{=} J_1\sigma \vert J_2\sigma$	$x\langle\tilde{v}\rangle\sigma$	$\stackrel{\text{def}}{=} (x\sigma)\langle\tilde{u}\rangle$	$(\tilde{u}_i = \tilde{v}_i\sigma)$
$D:$	$(D_1 \wedge D_2)\sigma$	$\stackrel{\text{def}}{=} D_1\sigma \wedge D_2\sigma$	$(J \triangleright P)\sigma$	$\stackrel{\text{def}}{=} J\sigma \triangleright P\sigma$	
	$a[D : P]\sigma$	$\stackrel{\text{def}}{=} a\sigma[D\sigma : P\sigma]$	$\mathtt{T}\sigma$	$\stackrel{\text{def}}{=} \mathtt{T}$	
	$\epsilon a[D : P : \sigma_2]\sigma$	$\stackrel{\text{def}}{=} \epsilon a\sigma[D\sigma : P\sigma : \sigma_2]$	$(\grave{}a)\sigma$	$\stackrel{\text{def}}{=} \grave{}(a\sigma)$	
$P:$	$(P_1 \vert P_2)\sigma$	$\stackrel{\text{def}}{=} P_1\sigma \mid P_2\sigma$	$\mathtt{0}\sigma$	$\stackrel{\text{def}}{=} \mathtt{0}$	
	$(\mathtt{def}\ D\ \mathtt{in}\ P)\sigma$	$\stackrel{\text{def}}{=} \mathtt{def}\ D\sigma\ \mathtt{in}\ P\sigma$	$\kappa\langle\tilde{v}\rangle\sigma$	$\stackrel{\text{def}}{=} (\kappa\sigma)\langle\tilde{u}\rangle$	$(\tilde{u}_i = \tilde{v}_i\sigma)$
	$\mathtt{gocore}\langle\alpha,\kappa\rangle\sigma$	$\stackrel{\text{def}}{=} \mathtt{gocore}\langle\alpha\sigma,\kappa\sigma\rangle$	$\mathtt{go}\langle\alpha,\kappa\rangle\sigma$	$\stackrel{\text{def}}{=} \mathtt{go}\langle\alpha\sigma,\kappa\sigma\rangle$	
	$\mathtt{gostate}\langle\alpha,\kappa\rangle\sigma$	$\stackrel{\text{def}}{=} \mathtt{gostate}\langle\alpha\sigma,\kappa\sigma\rangle$	$\mathtt{subst}\langle a\rangle\sigma$	$\stackrel{\text{def}}{=} \mathtt{subst}\langle a\sigma\rangle$	
	$\mathtt{fail}\langle\alpha,\kappa\rangle\sigma$	$\stackrel{\text{def}}{=} \mathtt{fail}\langle\alpha\sigma,\kappa\sigma\rangle$	$\mathtt{halt}\langle\rangle\sigma$	$\stackrel{\text{def}}{=} \mathtt{halt}\langle\rangle$	

Table 7.1: Definition of application of a substitution to a process, a definition, a join-pattern or a value.

7.2.2 Substitution of By-Type Values

The substitution functions $g_1(\cdot, \sigma)$ and $g_2(\cdot, \sigma)$ are used when the substitution σ only should be applied to certain parts of a process or definition.

The $g_1(., \sigma)$-function applies the substitution σ to the process or definition in its first argument. By-type references to locations will not be replaced but otherwise every occurrence of the names in $dom(\sigma)$ will be replaced. The function is defined in table 7.2.

The $g_2(P, \sigma)$-function only applies the substitution σ to location names marked as by-type in the process P. The g_2 function will only apply the substitution to values in pending messages and primitives at the "top-level" of P, i.e. not to processes inside definitions. The function is defined in table 7.3.

$$
\begin{array}{lllll}
v: & g_1(\grave{}a,\sigma) & \stackrel{\mathrm{def}}{=} \grave{}a & g_1(a,\sigma) & \stackrel{\mathrm{def}}{=} a\sigma \\
 & g_1(\alpha.x,\sigma) & \stackrel{\mathrm{def}}{=} g_1(\alpha,\sigma).x & g_1(x,\sigma) & \stackrel{\mathrm{def}}{=} x \\[1ex]
D: & g_1(D_1 \wedge D_2,\sigma) & \stackrel{\mathrm{def}}{=} g_1(D_1,\sigma) \wedge g_1(D_2,\sigma) & g_1(\mathtt{T},\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{T} \\
 & g_1(J \rhd P,\sigma) & \stackrel{\mathrm{def}}{=} J \rhd g_1(P,\sigma) & g_1(\grave{}a,\sigma) & \stackrel{\mathrm{def}}{=} \grave{}a \\
 & g_1(a[D:P],\sigma) & \stackrel{\mathrm{def}}{=} a\sigma[g_1(D,\sigma) : g_1(P,\sigma)] & g_1(a,\sigma) & \stackrel{\mathrm{def}}{=} a\sigma \\
 & g_1(\epsilon a[D:P:\sigma_2],\sigma) & \stackrel{\mathrm{def}}{=} \epsilon a\sigma[g_1(D,\sigma) : g_1(P,\sigma) : \sigma_2] & & \\[1ex]
P: & g_1(P_1|P_2,\sigma) & \stackrel{\mathrm{def}}{=} g_1(P_1,\sigma) \,|\, g_1(P_2,\sigma) & g_1(\mathtt{0},\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{0} \\
 & g_1(\mathtt{def}\ D\ \mathtt{in}\ P,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{def}\ g_1(D,\sigma)\ \mathtt{in}\ g_1(P,\sigma) & & \\
 & g_1(\kappa\langle\tilde{v}\rangle,\sigma) & \stackrel{\mathrm{def}}{=} g_1(\kappa,\sigma)\langle\tilde{u}\rangle \qquad (\tilde{u}_i = g_1(\tilde{v}_i,\sigma)) & & \\
 & g_1(\mathtt{go}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{go}\langle g_1(\alpha,\sigma), g_1(\kappa,\sigma)\rangle & & \\
 & g_1(\mathtt{gocore}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{gocore}\langle g_1(\alpha,\sigma), g_1(\kappa,\sigma)\rangle & & \\
 & g_1(\mathtt{gostate}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{gostate}\langle g_1(\alpha,\sigma), g_1(\kappa,\sigma)\rangle & & \\
 & g_1(\mathtt{subst}\langle a\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{subst}\langle a\sigma\rangle & & \\
 & g_1(\mathtt{fail}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{fail}\langle g_1(\alpha,\sigma), g_1(\kappa,\sigma)\rangle & & \\
 & g_1(\mathtt{halt}\langle\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{halt}\langle\rangle & &
\end{array}
$$

Table 7.2: Definition of the substitution function $g_1(\cdot,\sigma)$.

$$
\begin{array}{lllll}
v: & g_2(\grave{}a,\sigma) & \stackrel{\mathrm{def}}{=} a\sigma & g_2(a,\sigma) & \stackrel{\mathrm{def}}{=} a \\
 & g_2(\alpha.x,\sigma) & \stackrel{\mathrm{def}}{=} g_2(\alpha,\sigma).x & g_2(x,\sigma) & \stackrel{\mathrm{def}}{=} x \\[1ex]
P: & g_2(P_1|P_2,\sigma) & \stackrel{\mathrm{def}}{=} g_2(P_1,\sigma) \,|\, g_2(P_2,\sigma) & g_2(\mathtt{0},\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{0} \\
 & g_2(\mathtt{def}\ D\ \mathtt{in}\ P,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{def}\ D\ \mathtt{in}\ g_2(P,\sigma) & & \\
 & g_2(\kappa\langle\tilde{v}\rangle,\sigma) & \stackrel{\mathrm{def}}{=} g_2(\kappa,\sigma)\langle\tilde{u}\rangle \qquad (\tilde{u}_i = g_2(\tilde{v}_i,\sigma)) & & \\
 & g_2(\mathtt{go}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{go}\langle g_2(\alpha,\sigma), g_2(\kappa,\sigma)\rangle & & \\
 & g_2(\mathtt{gocore}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{gocore}\langle g_2(\alpha,\sigma), g_2(\kappa,\sigma)\rangle & & \\
 & g_2(\mathtt{gostate}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{gostate}\langle g_2(\alpha,\sigma), g_2(\kappa,\sigma)\rangle & & \\
 & g_2(\mathtt{subst}\langle a\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{subst}\langle a\rangle & & \\
 & g_2(\mathtt{fail}\langle\alpha,\kappa\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{fail}\langle g_2(\alpha,\sigma), g_2(\kappa,\sigma)\rangle & & \\
 & g_2(\mathtt{halt}\langle\rangle,\sigma) & \stackrel{\mathrm{def}}{=} \mathtt{halt}\langle\rangle & &
\end{array}
$$

Table 7.3: Definition of the substitution function $g_2(\cdot,\sigma)$.

7.2.3 Scoping

The sets of free, defined, and received names in a process or definition are defined in table 7.4. These definitions will be used to define the scope of names.

$dv[\cdot]$ is the set of defined variables, both location names and port names, whereas $dl[\cdot]$ is the set of defined location names. A free name is a name that hasn't been defined in a local definition. $fv[\cdot]$ is the set of free variables, both location and port names, whereas $fl[\cdot]$ is the set of free location names. The received variables, $rv[\cdot]$, in a join pattern are the formal names used to denote the values in a message to the defined port.

In the definitions in table 7.4, $S_1 \cup S_2$ will denote union of any two sets, whereas $S_1 \uplus S_2$ will denote the union of two disjoint sets.

The definition of $fv[J \triangleright P]$ has been changed from the definition in the DRCHAM. This is because $rv[J]$ need to be considered free variables because the names in $rv[J]$ shouldn't appear as location names elsewhere. That could cause the application of the substitution function $g_1(\cdot, \sigma)$ to a reduction definition $J \triangleright P$ to remove the connection between the received variables in J and in P.

7.3 Semantics

The semantics of the language is given by a chemical abstract machine, the MLCHAM, where the molecules are constituted by the terms of the Mobility Language.

The chemical rules of the MLCHAM work on multisets of labelled solutions. A labelled solution has the form $\mathcal{D} \vdash_{\phi}^{\sigma} \mathcal{P}$ and it represents a location. The four parts of a labelled solution are:

- a multiset $\mathcal{D}$ of definitions. When a location is understood as a component, the multiset $\mathcal{D}$ can be seen as the code component.
- a multiset $\mathcal{P}$ of processes, i.e. pending messages, local definitions, or primitives. When the location is understood as a component, the multiset $\mathcal{P}$ represents the local state of the component and the control information of the computation.
- a string of location names ϕ. This string holds the name of the location and all its parent locations, starting from the top-level location and ending with the location itself.
- a substitution σ from general location (component) names to instances of the locations.

A multiset of labelled solutions is written as the labelled solutions separated by "$||$". A location $\vdash_{\phi}$ is a sublocation of another location $\vdash_{\psi}$ if ψ is a prefix of ϕ.

A system which is described by the process P in the Mobility Language will be represented by the labelled solution $\emptyset \vdash^{\{\}} \{\!|P|\!\}$.

$$
\begin{array}{lll@{\qquad}ll}
v: & fv[a] \stackrel{\text{def}}{=} \{a\} & & fv[\text{`}a] & \stackrel{\text{def}}{=} \{a\} \\
 & fv[a.x] \stackrel{\text{def}}{=} \{a, x\} & & fv[\text{`}a.x] & \stackrel{\text{def}}{=} \{a, x\} \\
 & fv[x] \stackrel{\text{def}}{=} \{x\} & & & \\
 & & & & \\
 & fl[a] \stackrel{\text{def}}{=} \{a\} & & fl[\text{`}a] & \stackrel{\text{def}}{=} \{a\} \\
 & fl[a.x] \stackrel{\text{def}}{=} \{a\} & & fl[\text{`}a.x] & \stackrel{\text{def}}{=} \{a\} \\
 & fl[x] \stackrel{\text{def}}{=} \emptyset & & &
\end{array}
$$

$$
\begin{array}{llll@{\qquad}ll}
J: & dv[J_1|J_2] & \stackrel{\text{def}}{=} & dv[J_1] \cup dv[J_2] & dv[x\langle\tilde{v}\rangle] & \stackrel{\text{def}}{=} \{x\} \\
 & rv[J_1|J_2] & \stackrel{\text{def}}{=} & rv[J_1] \uplus rv[J_2] & rv[x\langle\tilde{v}\rangle] & \stackrel{\text{def}}{=} \{u : u \in \tilde{v}\}
\end{array}
$$

$$
\begin{array}{llll@{\qquad}ll}
D: & fv[D_1 \wedge D_2] & \stackrel{\text{def}}{=} & fv[D_1] \cup fv[D_2] & fv[\mathtt{T}] & \stackrel{\text{def}}{=} \emptyset \\
 & fv[J \rhd P] & \stackrel{\text{def}}{=} & dv[J] \cup fv[P] & fv[a] & \stackrel{\text{def}}{=} \{a\} \\
 & fv[a[D : P]] & \stackrel{\text{def}}{=} & \{a\} \cup fv[D] \cup fv[P] & fv[a'] & \stackrel{\text{def}}{=} \{a\} \\
 & fv[a[D : P : \sigma]] & \stackrel{\text{def}}{=} & \{a\} \cup fv[D] \cup fv[P] \cup range(\sigma) & & \\
 & & & & & \\
 & fl[D_1 \wedge D_2] & \stackrel{\text{def}}{=} & fl[D_1] \cup fl[D_2] & fl[\mathtt{T}] & \stackrel{\text{def}}{=} \emptyset \\
 & fl[J \rhd P] & \stackrel{\text{def}}{=} & fl[P] & fl[a] & \stackrel{\text{def}}{=} \{a\} \\
 & fl[a[D : P]] & \stackrel{\text{def}}{=} & \{a\} \cup fl[D] \cup fl[P] & fl[a'] & \stackrel{\text{def}}{=} \{a\} \\
 & fl[a[D : P : \sigma]] & \stackrel{\text{def}}{=} & \{a\} \cup fl[D] \cup fl[P] \cup range(\sigma) & & \\
 & & & & & \\
 & dv[D_1 \wedge D_2] & \stackrel{\text{def}}{=} & dv[D_1] \cup dv[D_2] & dv[J \rhd P] & \stackrel{\text{def}}{=} dv[J] \\
 & dv[a[D : P]] & \stackrel{\text{def}}{=} & \{a\} \uplus dv[D] & dv[\mathtt{T}] & \stackrel{\text{def}}{=} \emptyset \\
 & dv[a[D : P : \sigma]] & \stackrel{\text{def}}{=} & \{a\} \uplus dv[D] & dv[\alpha] & \stackrel{\text{def}}{=} \emptyset \\
 & & & & & \\
 & dl[D_1 \wedge D_2] & \stackrel{\text{def}}{=} & dl[D_1] \uplus dl[D_2] & dl[J \rhd P] & \stackrel{\text{def}}{=} \emptyset \\
 & dl[a[D : P]] & \stackrel{\text{def}}{=} & \{a\} \uplus dl[D] & dl[\mathtt{T}] & \stackrel{\text{def}}{=} \emptyset \\
 & dl[a[D : P : \sigma]] & \stackrel{\text{def}}{=} & \{a\} \uplus dl[D] & dl[\alpha] & \stackrel{\text{def}}{=} \emptyset
\end{array}
$$

$$
\begin{array}{llll@{\qquad}ll}
P: & fv[P_1|P_2] & \stackrel{\text{def}}{=} & fv[P_1] \cup fv[P_2] & fv[\kappa\langle\tilde{v}\rangle] & \stackrel{\text{def}}{=} (\bigcup_{u \in \tilde{v}} fv[u]) \cup fv[\kappa] \\
 & fv[\mathtt{def}\ D\ \mathtt{in}\ P] & \stackrel{\text{def}}{=} & (fv[P] \cup fv[D]) - dv[D] & fv[\mathtt{0}] & \stackrel{\text{def}}{=} \emptyset \\
 & fv[\mathtt{gocore}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} & fv[\alpha] \cup fv[\kappa] & fv[\mathtt{go}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} fv[\alpha] \cup fv[\kappa] \\
 & fv[\mathtt{gostate}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} & fv[\alpha] \cup fv[\kappa] & fv[\mathtt{subst}\langle a\rangle] & \stackrel{\text{def}}{=} fv[a] \\
 & fv[\mathtt{fail}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} & fv[\alpha] \cup fv[\kappa] & fv[\mathtt{halt}\langle\rangle] & \stackrel{\text{def}}{=} \emptyset \\
 & & & & & \\
 & fl[P_1|P_2] & \stackrel{\text{def}}{=} & fl[P_1] \cup fl[P_2] & fl[\kappa\langle\tilde{v}\rangle] & \stackrel{\text{def}}{=} (\bigcup_{u \in \tilde{v}} fl[u]) \cup fl[\kappa] \\
 & fl[\mathtt{def}\ D\ \mathtt{in}\ P] & \stackrel{\text{def}}{=} & (fl[P] \cup fl[D]) - dl[D] & fl[\mathtt{0}] & \stackrel{\text{def}}{=} \emptyset \\
 & fl[\mathtt{gocore}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} & fl[\alpha] \cup fl[\kappa] & fl[\mathtt{go}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} fl[\alpha] \cup fl[\kappa] \\
 & fl[\mathtt{gostate}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} & fl[\alpha] \cup fl[\kappa] & fl[\mathtt{subst}\langle a\rangle] & \stackrel{\text{def}}{=} fl[a] \\
 & fl[\mathtt{fail}\langle\alpha,\kappa\rangle] & \stackrel{\text{def}}{=} & fl[\alpha] \cup fl[\kappa] & fl[\mathtt{halt}\langle\rangle] & \stackrel{\text{def}}{=} \emptyset
\end{array}
$$

Table 7.4: Definitions of the free variables, $fv[\cdot]$, the free location names, $fl[\cdot]$, the defined variables, $dv[\cdot]$, the defined locations, $dl[\cdot]$, and the received variables, $rv[\cdot]$, in a process, definition or join-pattern.

A multiset of labelled solutions will evolve according to a set of heating/cooling rules ($\rightleftharpoons$) and a set of reaction rules ($\rightarrow$). The application of a heating/cooling rule in a solution corresponds to a structural rearrangement of the processes or definitions. Such a structural rearrangement can allow reaction rules to be applied to the multiset of solutions. The application of a reaction rule in a solution corresponds to a basic computation step in the system which is represented by the solution.

The heating/cooling rules for the MLCHAM are given in table 7.6 and table 7.7. The reaction rules are given in table 7.5. The rules will only show the parts of the multiset of labelled solutions which are involved in the rule, i.e. the rules assume an implicit context. The side conditions of certain rules will also involve the entire context of the rule. When a rule involves two solutions, $\vdash_\phi \parallel \vdash_\psi$, the rule can also be applied to a multiset of solutions where $\vdash_\phi$ and $\vdash_\psi$ are the same solution. E.g. the rule *Comm* can also be applied when a location sends a message to itself, i.e. when $\phi = \psi a$.

Red	$J \rhd P \vdash^\sigma_\phi J\sigma_{rv} \quad \rightarrow \quad J \rhd P \vdash^\sigma_\phi g_2(P\sigma_{rv}, \sigma)$	$(dom(\sigma_{rv}) = rv[J]$, ϕ alive)
Comm	$\vdash_\phi a.x\langle\tilde{v}\rangle \parallel \vdash_{\psi\epsilon a} \quad \rightarrow \quad \vdash_\phi \parallel \vdash_{\psi\epsilon a} x\langle\tilde{v}\rangle$	(ϕ alive)
Halt	$a[D : P \vert \mathtt{halt}\langle\rangle : \sigma] \vdash_\phi \quad \rightarrow \quad \Omega a[D : P : \sigma] \vdash_\phi$	(ϕ alive)
Detect	$\vdash^\sigma_\phi \mathtt{fail}\langle\alpha, \kappa\rangle \parallel \vdash_{\psi\epsilon b} \quad \rightarrow \quad \vdash^\sigma_\phi \kappa\langle\rangle \parallel \vdash_{\psi\epsilon b}$	$(g_2(\alpha, \sigma) = b$, $\psi\epsilon b$ dead, ϕ alive)
Move	$a_1[\,c[D_2 : P_2 \vert \mathtt{go}\langle a_2, \kappa\rangle : \sigma_2] \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_\phi \parallel \epsilon a_2[D_3 : P_3 : \sigma_3] \vdash_\psi \quad \rightarrow$ $a_1[D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_\phi \parallel \epsilon a_2[\,c[\,D_2 : P_2 \vert \kappa\langle\rangle : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash_\psi$ (D_2 is fully expanded, $c \notin range(\sigma_4\sigma_1)$, for all $b \in dl[D_2]$: $b \notin range(\sigma_4\sigma_1)$, ϕ alive)	
MoveC	$a_1[\,c[D_2 : P_2 \vert \mathtt{gocore}\langle a_2, \kappa\rangle : \sigma_2] \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_\phi \parallel \epsilon a_2[D_3 : P_3 : \sigma_3] \vdash_\psi \quad \rightarrow$ $a_1[D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_\phi \parallel \epsilon a_2[\,c[\,D_2 : P_2 \vert \kappa\langle\rangle : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash_\psi$ (c frozen, $c \notin range(\sigma_4\sigma_1)$, for all $b \in dl[D_2]$: $b \notin range(\sigma_4\sigma_1)$, ϕ alive)	
MoveS	$a_1[\,c[\beta : P_2 \vert \mathtt{gostate}\langle a_2, \kappa\rangle : \sigma_2] \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_\phi \parallel \epsilon a_2[D_3 : P_3 : \sigma_3] \vdash_\psi \quad \rightarrow$ $a_1[D_1 : P_1 : \sigma_1] \vdash^{\sigma_4}_\phi \parallel \epsilon a_2[\,c[\,\beta : P_2 \vert \kappa\langle\rangle : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash_\psi$ ($c \notin range(\sigma_4\sigma_1)$, ϕ alive)	

Table 7.5: The reaction rules of the MLCHAM

Some of the side conditions on the rules require a little further explanation:

ϕ **dead** means that the string ϕ contains the marker Ω

ϕ **alive** means that ϕ does not contain the marker Ω.

D **is fully expanded** means that any location reference appearing in D has been expanded according to the *Copy* rule. Formally this can be expressed as:
$\neg\exists c \in \mathcal{L}, D_1, D_2 : (D = c) \vee (D = D_1 \wedge c) \vee (D = D_1 \wedge c \wedge D_2) \vee (D = c \wedge D_2)$

c **frozen** means that location c should be cooled as much as possible, particularly using the *Contract* rule (the cooling according to the rules *Subst* and *SolDef* is already covered by other side conditions). This means that a location will bring as few definitions as possible when it moves using the `gocore` primitive.

range(σ) **fresh** means that all names in the range of the substitution not are free variables in the constructs of the rule or in the context of the rule. In the *Def* rule, the side condition "$range(\sigma_{dv})$ fresh" will mean that:

$$range(\sigma_{dv}) \subseteq \mathcal{N} \cup \mathcal{L} - fv[context] - fv[\texttt{def}\ D\ \texttt{in}\ P]$$

where "*context*" refers to all the parts of the multiset of solutions that aren't mentioned in the rule. In the *Copy* and *Contract* rules the side condition "$range(\sigma_{dl})$ fresh" will mean that:

$$range(\sigma_{dl}) \subseteq \mathcal{L} - fl[context] - fl[\ a[D_1 : P_1]\ ] - fl[\ c[D_2 : P_2]\ ]$$

no sublocation of a **is on the form** $b[D_1 : P_1|\texttt{subst}\langle c\rangle\ ]$ means that no immediate sublocation of a contains a `subst` primitive in its process part.

no sublocation of a **is in solution form** means that there is no solution $\vdash_{\phi\epsilon a\psi}$ where ψ is a non-empty label. The condition ensures that whenever a location is in definition form, it will include all its sublocations.

$$
\begin{array}{lrcl}
Join & \vdash P_1|P_2 & \rightleftharpoons & \vdash P_1, P_2 \\
Null & \vdash \mathtt{0} & \rightleftharpoons & \vdash \\
Def & \vdash \mathtt{def}\ D\ \mathtt{in}\ P & \rightleftharpoons & D\sigma_{dv} \vdash P\sigma_{dv} \\
 & \multicolumn{3}{c}{(dom(\sigma_{dv}) = dv[D],\ range(\sigma_{dv}) \text{ fresh})} \\
DefAnd & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash \\
DComm: & \epsilon a[D_1 \wedge D_2 : P : \sigma] \vdash & \rightleftharpoons & \epsilon a[D_2 \wedge D_1 : P : \sigma] \vdash \\
DAssoc: & \epsilon a[(D_1 \wedge D_2) \wedge D_3 : P : \sigma] \vdash & \rightleftharpoons & \epsilon a[D_1 \wedge (D_2 \wedge D_3) : P : \sigma] \vdash \\
DComm2: & \multicolumn{3}{c}{\epsilon b[\ \epsilon a[D_1 \wedge D_2 : P : \sigma]\ \wedge D_3 : P_3 : \sigma_3\ ] \vdash\ \ \rightleftharpoons} \\
 & \multicolumn{3}{c}{\epsilon b[\ \epsilon a[D_2 \wedge D_1 : P : \sigma]\ \wedge D_3 : P_3 : \sigma_3\ ] \vdash} \\
DAssoc2: & \multicolumn{3}{c}{\epsilon b[\ \epsilon a[(D_1 \wedge D_2) \wedge D_3 : P : \sigma]\ \wedge D_4 : P_4 : \sigma_4\ ] \vdash\ \ \rightleftharpoons} \\
 & \multicolumn{3}{c}{\epsilon b[\ \epsilon a[D_1 \wedge (D_2 \wedge D_3) : P : \sigma]\ \wedge D_4 : P_4 : \sigma_4\ ] \vdash} \\
DefNull & \mathtt{T} \vdash & \rightleftharpoons & \vdash \\
DefNull2: & \epsilon a[D \wedge \mathtt{T} : P : \sigma] \vdash & \rightleftharpoons & \epsilon a[D : P : \sigma] \vdash \\
DefNull3: & \multicolumn{3}{c}{\epsilon b[\ \epsilon a[D \wedge \mathtt{T} : P : \sigma]\ \wedge D_2 : P_2 : \sigma_2\ ] \vdash\ \ \rightleftharpoons} \\
 & \multicolumn{3}{c}{\epsilon b[\ \epsilon a[D : P : \sigma]\ \wedge D_2 : P_2 : \sigma_2\ ] \vdash} \\
DefLoc & a[D : P] \vdash & \rightharpoonup & a[D : P : \{\}] \vdash \\
DefLoc2 & \epsilon b[\ a[D : P] \wedge D_2 : P_2 : \sigma] \vdash & \rightharpoonup & \epsilon b[\ a[D : P : \{\}] \wedge D_2 : P_2 : \sigma] \vdash \\
DefSol & \epsilon a[D : P : \sigma_2] \vdash^{\sigma_1}_{\phi} & \rightharpoonup & \vdash^{\sigma_1}_{\phi}\ \ \|\ \{\!|D|\!\} \vdash^{\sigma_1\sigma_2}_{\phi\epsilon a} \{\!|P|\!\} \\
 & \multicolumn{3}{c}{(D \text{ is fully expanded, no sublocation of } a \text{ is on the form } b[D_1 : P_1|subst\langle c\rangle])} \\
SolDef & \epsilon a[D : P : \sigma_2] \vdash^{\sigma_1}_{\phi} & \leftharpoondown & \vdash^{\sigma_1}_{\phi}\ \ \|\ \{\!|D|\!\} \vdash^{\sigma_1\sigma_2}_{\phi\epsilon a} \{\!|P|\!\} \\
 & \multicolumn{3}{c}{(\text{no sublocation of } a \text{ is in solution form})}
\end{array}
$$

Table 7.6: The heating/cooling rules that allow structural rearrangement of the definitions and processes in a location.

Copy	$\epsilon a[\,\beta \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_3}_{\phi} \parallel c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \quad \rightharpoonup$ $\epsilon a[\,g_1(D_2 \wedge D_1\,,\,\sigma_2\sigma_{dl}) : g_1(P_2 \vert P_1\,,\,\sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \vdash^{\sigma_3}_{\phi}$ $\parallel c[D_2 : P_2 : \sigma_2] \vdash_{\psi}$ ($g_2(\beta, \sigma_3\sigma_1) = c$, $\beta \neq$ \`c, $dom(\sigma_{dl}) = dl[D_2]$, $range(\sigma_{dl})$ fresh, ψ alive)
*Copy*2	$\epsilon d[\,\epsilon a[\beta \wedge D_1 : P_1 : \sigma_1] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi} \parallel c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \quad \rightharpoonup$ $\epsilon d[\,\epsilon a[\,g_1(D_2 \wedge D_1\,,\,\sigma_2\sigma_{dl}) : g_1(P_2 \vert P_1\,,\,\sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi}$ $\parallel c[D_2 : P_2 : \sigma_2] \vdash_{\psi}$ ($g_2(\beta, \sigma_4\sigma_3\sigma_1) = c$, $\beta \neq$ \`c, $dom(\sigma_{dl}) = dl[D_2]$, $range(\sigma_{dl})$ fresh, ψ alive)
*Copy*3	$d[\,\epsilon a[\beta \wedge D_1 : P_1 : \sigma_1] \wedge c[D_2 : P_2 : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi} \quad \rightharpoonup$ $d[\,\epsilon a[\,g_1(D_2 \wedge D_1\,,\,\sigma_2\sigma_{dl}) : g_1(P_2 \vert P_1\,,\,\sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \wedge$ $c[D_2 : P_2 : \sigma_2] \wedge D_3 : P_3 : \sigma_3] \vdash^{\sigma_4}_{\phi}$ ($g_2(\beta, \sigma_4\sigma_3\sigma_1) = c$, $\beta \neq$ \`c, $dom(\sigma_{dl}) = dl[D_2]$, $range(\sigma_{dl})$ fresh, ϕ alive)
Contract	$\epsilon a[$\`$b \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_3}_{\phi} \parallel c[D_2 : \mathtt{0} : \sigma_2] \vdash^{\sigma_4}_{\psi} \quad \leftharpoondown$ $\epsilon a[\,g_1(D_2 \wedge D_1\,,\,\sigma_2\sigma_{dl}) : g_1(P_1\,,\,\sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \vdash^{\sigma_3}_{\phi}$ $\parallel c[D_2 : \mathtt{0} : \sigma_2] \vdash^{\sigma_4}_{\psi}$ ($g_2($\`$b, \sigma_3\sigma_1) = c$, $b \notin range(\sigma_3\sigma_1)$, $b \notin range(\sigma_4)$, $dom(\sigma_{dl}) = dl[D_2]$, $range(\sigma_{dl})$ fresh, $range(\sigma_2)$ fresh in a, ψ alive)
Subst	$\epsilon a[\,\epsilon c[\,D_2 : P_2 \vert \mathtt{subst}\langle b\rangle : \sigma_2\,] \wedge D_1 : P_1 : \sigma_1\,] \quad \vdash^{\sigma_3} \quad \rightleftharpoons$ $\epsilon a[\,\epsilon c[\,g_1(D_2, \{c/b\}) : g_1(P_2, \{c/b\}) : \sigma_2\,] \wedge$ $g_1(D_1, \{c/b\}\sigma_2) : g_1(P_1, \{c/b\}\sigma_2) : \sigma_1\{c/b\}\sigma_2\,] \quad \vdash^{\sigma_3}$ (D_2 is fully expanded, $(\{c\} \cup dl[D_2]) \cap range(\sigma_3\sigma_1) = \emptyset$, $(\{c\} \cup range(\sigma_2)) \cap (fl[D_1] \cup fl[P_1]) = \emptyset$)

Table 7.7: The heating/cooling rules that support copying of definitions and introduction of component instances.

Chapter 8

Examples

The Mobility Language has been designed to allow one to model the different mechanisms used in the technologies described in chapter 2. This chapter will give examples that demonstrate how some of the mechanisms can be modelled. For a couple of the examples, it will also be shown how the chemical rules from the MLCHAM can be applied to allow the modelled computations to be carried out.

The first example will show how a Java object can move from one JVM to another, bringing only the state and not the class. The next example models the dynamic class downloading in Java. The computations for both of these examples will be carried out in order to demonstrate the use of all the basic chemical rules. Section 8.3 will combine the preceding examples to a larger model describing an RMI call. Finally, section 8.4 will show how remote cloning of an agent can be modelled.

8.1 Object Mobility in Java

When a Java object moves between JVMs, it will only be the object's state that moves. When the object is going to be used for computations, a local class definition will be used. The following example shows how this can be modelled by modelling an object as a location having a by-type binding to its class definition and by moving the location using the `gostate` primitive.

A library will contain definitions of the classes c_1, c_2, and c_3. Every JVM will have a copy of this library and will thus have a local instance of every class. An object of class c_2 is located at jvm_1 but the object will be moved to jvm_2 where the computations can continue:

$$\begin{array}{ll} \texttt{def} & library[\ c_1[D_1 : \mathsf{0}] \wedge \\ & \qquad\qquad c_2[move\langle l\rangle \triangleright \texttt{gostate}\langle l,f\rangle \wedge f\langle\rangle | state\langle s\rangle \triangleright P : \mathsf{0}] \wedge \\ & \qquad\qquad c_3[D_3 : \mathsf{0}] : \mathsf{0}\ ] \\ \wedge & jvm_1[\quad library \wedge obj[`c_2 : state\langle s_0\rangle] : \mathsf{0}\ ] \\ \wedge & jvm_2[\quad library : \mathsf{0}\ ] \\ \texttt{in} & obj.move\langle jvm_2\rangle \end{array}$$

The location *obj* is moved by sending it a *move* message. This is done to illustrate how the definitions from c_2 can first be copied to the location in order to allow computations to be performed, after which the definitions can be contracted again to allow the location to move using the `gostate` primitive. After the movement, the location can be expanded again using jvm_2's instance of c_2, and the computation of operation f can continue.

The development of a solution containing the above process, and a detailed description of the necessary computation steps, can be seen in appendix D.1.

8.2 Dynamic Class Downloading

A Java object can be moved to a JVM which doesn't have a local instance of the object's class. This could be the case for RMI stub objects or objects passed as argument to an RMI call where the object is an instance of a subclass of a standard class. The JVM's class loader will in this case take care of dynamically downloading the class definition from an HTTP server. The following example will show how the dynamic downloading of a class can be modelled, using the `go` primitive, the `subst` primitive, and copying of a location.

A JVM holds an object of a class c_2 which isn't a class in the JVM's standard library. The JVM will therefore ask the class loader to fetch the class from its codebase. An HTTP server will hold a copy of all such classes that aren't classes in Java's standard library, including the class c_2. The server will accept requests for a class, upon which it will create a copy of this class and ship it off to the requesting JVM:

$$\begin{array}{ll} \texttt{def} & ext\ [\ c_1[D_1 : \mathsf{0}] \wedge c_2[D_2 : \mathsf{0}] \wedge c_3[D_3 : \mathsf{0}] \wedge \ldots : \mathsf{0}\ ] \\ \wedge & http\ [\ ext \wedge \\ & \qquad fetch\langle class, l\rangle \triangleright \texttt{def}\ \ newc[\ `class : \texttt{subst}\langle class\rangle \,|\, \texttt{go}(l); \mathsf{0}] \\ & \qquad\qquad\qquad\qquad \texttt{in}\ \mathsf{0} \\ & \quad : \mathsf{0}] \\ \wedge & jvm\ [\ classloader[\ get\langle class, cb\rangle \triangleright cb.fetch\langle class, jvm\rangle\ ] \wedge \\ & \qquad obj[\ `c_2 : state\langle s\rangle] \\ & \quad : classloader.get\langle c_2, http\rangle\ ] \\ \texttt{in} & \quad \mathsf{0} \end{array}$$

When the HTTP server creates a copy of the requested class, it will create the copy from its local instance of the requested class. That is why the reference to *class* in *newc* is marked as by-type.

The copy of the class will be moved using the `go` primitive, which will ensure that all definitions from the original class has been copied to *newc* before it moves. When the copy of the requested class arrives at the JVM, the class will tell the JVM that it is an instance of the requested class, in this case an instance of c_2.

The development of a solution containing the above process, and a detailed description of the necessary computation steps, can be seen in appendix D.2.

8.3 RMI

Object mobility and dynamic class downloading are the mechanisms that are used to support RMI. The stub of a remote server object can be fetched from the remote RMIregistry, and the stub class can be fetched from an HTTP server. The full stub object can then be used to invoke operations in the remote server object.

Figure 8.1 shows a model of an RMI call. A JVM, jvm_1, hosts a server object, which offers a set of operations. A stub for the server object is created and registered with the RMIregistry running on the same JVM. Another JVM, jvm_2, hosts a client object, which wants to use the operations offered by the server object. The client obtains a stub to the server object by fetching the stub from the RMIregistry in jvm_2. The client then fetches the stub's class definition from an HTTP server. The client can now invoke the operations on the stub, which will forward the call to the remote server object.

The model starts by defining the class for the server object, *ServerClass*, and the class for the stub object, *StubClass*. The server is a location with two defined operations, $m1(arg)$ and $m_2()$. The stub class has the same interface and will also offer the operations $m_1(arg)$ and $m_2()$. A stub object will know how to contact the corresponding server object, and the operations in the stub class will simply forward a call to the server object, wait for a reply, and then return the result to the client.

The stub class defines one more operation, *move*. The operation will be used when a stub object should move to another JVM. The stub will be moved without the definitions from *StubClass*, but a message will be sent to the destination JVM with information about how to obtain the *StubClass*.

The location JVM models a general JVM and it will later be used to create concrete JVMs. Every JVM has an RMIregistry, *Registry*, which keeps a register over the server objects at the JVM. The registry makes use of two internal operations, *register* and *get*. These operations should respectively register a given stub under a given name and find the stub registered under a given name.

A server object registers its stub with the registry using the *bind* operation. This will result in the stub object being moved to the registry and being registered under a given name. A client object can obtain a stub object registered under a given name by using the *lookup* operation. This will result in a copy of the stub object being moved to the JVM of the client object.

$$
\begin{array}{l}
\mathit{ServerClass}[\ m_1(\mathit{args}) \rhd \ldots\ \texttt{reply}\ x\ \texttt{to}\ m_1\ \wedge \\
\qquad\qquad\quad m_2() \rhd \ldots\ \texttt{reply to}\ m_2 \\
\qquad\qquad\ :\mathtt{0}\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathit{StubClass}[\ m_1\langle \mathit{args}, \kappa\rangle | \mathit{server}\langle s\rangle \rhd \mathit{server}\langle s\rangle\ \mid\ \texttt{let}\ x = s.m_1(\mathit{args})\ \texttt{in}\ \kappa\langle x\rangle\ \wedge \\
\qquad\qquad m_2\langle \mathit{args}, \kappa\rangle | \mathit{server}\langle s\rangle \rhd \mathit{server}\langle s\rangle\ \mid\ s.m_2(\mathit{args}); \kappa\langle\rangle\ \wedge \\
\qquad\qquad \mathit{move}\langle l, \kappa\rangle | \mathit{codebase}\langle \mathit{cb}\rangle \rhd \mathit{codebase}\langle \mathit{cb}\rangle \mid \texttt{gostate}\langle l, \kappa\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid l.\mathit{loadClass}\langle \mathit{StubClass}, \mathit{cb}\rangle \\
\qquad\quad :\mathtt{0}\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathit{JVM}[\ \mathit{Registry}[\ \mathit{bind}\langle \mathit{name}, \mathit{stub}\rangle \rhd \mathit{stub}.\mathit{move}(\mathit{Registry}); \mathit{register}\langle \mathit{name}, \mathit{stub}\rangle\ \wedge \\
\qquad\qquad\qquad\quad \mathit{lookup}\langle \mathit{name}, l, \kappa\rangle \rhd \texttt{let}\ s = \mathit{get}(\mathit{name})\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{def}\ \mathit{stub}[s:\mathtt{0}]\ \texttt{in}\ \mathit{stub}.\mathit{move}(l); \kappa\langle \mathit{stub}\rangle\ \wedge \\
\qquad\qquad\qquad\quad \mathit{register}\langle \mathit{name}, \mathit{stub}\rangle \rhd \ldots\ \wedge \\
\qquad\qquad\qquad\quad \mathit{get}\langle \mathit{name}, \kappa\rangle \rhd \ldots\ \kappa\langle \mathit{stub}\rangle \\
\qquad\qquad\qquad :\mathtt{0}\ ] \\
\qquad \mathit{getRegistry}\langle\kappa\rangle \rhd \kappa\langle \mathit{Registry}\rangle\ \wedge \\
\qquad \mathit{loadClass}\langle c, \mathit{cb}\rangle | \mathit{jvm}\langle l\rangle \rhd \mathit{jvm}\langle l\rangle \mid \mathit{cb}.\mathit{fetch}\langle c, l\rangle \\
\quad :\mathtt{0}\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathit{http}[\ \mathit{stubc}[\mathit{StubClass} : \texttt{subst}\langle \mathit{StubClass}\rangle]\ \wedge \\
\qquad \mathit{fetch}\langle \mathit{class}, l\rangle \rhd \texttt{def}\ \mathit{newc}[\ \grave{}\mathit{class} : \texttt{subst}\langle \mathit{class}\rangle | \texttt{go}(l); \mathtt{0}\ ]\ \texttt{in}\ \mathtt{0} \\
\quad :\mathtt{0}\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathit{jvm}_1[\ \mathit{JVM}\ \wedge\ \mathit{stubc}_2[\mathit{StubClass} : \texttt{subst}\langle \mathit{StubClass}\rangle] \\
\qquad :\mathit{jvm}\langle \mathit{jvm}_1\rangle\ \mid \\
\qquad\ \texttt{def}\ \mathit{serverobj}[\mathit{ServerClass} : \mathtt{0}]\ \wedge \\
\qquad\qquad \mathit{stub}[\ \grave{}\mathit{StubClass} : \mathit{server}\langle \mathit{serverobj}\rangle \mid \mathit{codebase}\langle \mathit{http}\rangle\ ] \\
\qquad\ \texttt{in}\ \ \mathit{Registry}.\mathit{bind}\langle \text{“}\mathit{compute}'', \mathit{stub}\rangle\ ]
\end{array}
$$

$$
\begin{array}{l}
\mathit{jvm}_2[\ \mathit{JVM}\ \wedge \\
\qquad \mathit{client}[\ \mathtt{T} : \texttt{let}\ \mathit{reg} = \mathit{jvm}_1.\mathit{getRegistry}() \\
\qquad\qquad\qquad \texttt{in}\ \ \texttt{let}\ s = \mathit{reg}.\mathit{lookup}(\text{“}\mathit{compute}'', \mathit{jvm}_2) \\
\qquad\qquad\qquad\qquad s.m_2(); \mathtt{0}\ ] \\
\qquad :\mathit{jvm}\langle \mathit{jvm}_2\rangle\ ]
\end{array}
$$

Figure 8.1: A model of an RMI call from a client object at jvm_2 to a server object at jvm_1.

The *getRegistry* operation in a JVM will return a reference to the RMIregistry. This operation is necessary because every JVM will get its own name for the registry. The *loadClass* operation will fetch a given class from a given codebase. In a more detailed example, the *loadClass* operation should first check if the class is already loaded into the JVM.

The definitions from JVM are copied to the two JVMs, jvm_1 and jvm_2. Each JVM will keep its own name in an internal message, which is needed by the *loadClass* operation. The model also includes an HTTP server, *http*, which holds a local instance of *StubClass* and otherwise works the same as the server in the previous example.

The server class is instantiated to a server object located at jvm_1. A corresponding stub object will also be created and registered with the local RMIregistry under the name "compute". The stub object will know the name of the server object and its codebase will be set to the HTTP server.

A client object is located at jvm_2. The client will first obtain a reference to the RMIregistry at jvm_1. It can then obtain a stub for the server object by looking up in the registry under the name "compute". When the stub is moved to jvm_2, the JVM will also be told that it should fetch the *StubClass* from *http*. The client can now use the stub object to invoke operations in the server object, just as if it was a local object.

8.4 Remote Cloning

Many agent technologies, e.g. Aglets, allow a clone of an agent to be created in another location. None of the formal languages described in chapter 4 can model this, but it is possible with the Mobility Language. The following example demonstrates how this remote cloning can be modelled:

$$
\begin{array}{l}
\mathit{Site}[\ \mathit{AgentRuntime}[\ \mathit{clone}\langle a,l,\kappa\rangle \rhd \texttt{def}\ a_2[a : \texttt{go}\langle l, \mathit{AgentRuntime}.\kappa_2\rangle] \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \kappa_2\langle\rangle \rhd a.\kappa\langle a_2\rangle \ \mid\ a_2.\kappa\langle a\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{in}\ \ \mathsf{0} \\
\qquad\qquad\qquad\quad : \mathsf{0}\ ]\ \ : \mathsf{0}\ ]
\end{array}
$$

$$
\mathit{Agent}[\ \mathit{clone}\langle a,l,\kappa\rangle \rhd\ \grave{}\mathit{AgentRuntime}.\mathit{clone}\langle a,l,\kappa\rangle : \mathsf{0}\ ]
$$

$$
\mathit{site}_1[\ \mathit{Site} : \mathsf{0}\ ]
$$

$$
\begin{array}{l}
\mathit{site}_2[\ \mathit{Site}\ \wedge \\
\qquad \mathit{agent}_1[\ \mathit{Agent} : \texttt{let}\ \ \mathit{agent}_2 = \mathit{clone}(\mathit{agent}_1, \mathit{site}_1)\ \ \texttt{in}\ \ P \ldots\ ] \\
\quad : \mathsf{0}\ ]
\end{array}
$$

In the example, an agent runtime system will run at every site, and it will be this runtime system that takes care of the creation of a clone. Every agent will have a *clone* operation, but the operation will simply forward the call to the *clone* operation in the *AgentRuntime*. This operation will create a copy of the given agent and let the clone move to the given location. When the clone has moved to the location, it will receive the name of the original agent and can then continue the computations. The original agent will receive the name of the clone and can also continue with the same computations.

Chapter 9

Evaluation and Discussion

The examples in the preceding chapter have shown that the Mobility Language can be used to model the mobility mechanisms used in Java. However, not all of the posed requirements to the modelling capabilities of a language have been fulfilled. The following sections will look at which aspects of distributed systems with code mobility that the language can model, and further give suggestions to how the Mobility Language can be extended in order to meet more of the requirements. Some alternative approaches to how the modelling language could have been constructed will also be discussed, and possible future work in connection with the Mobility Language will be suggested.

9.1 Fulfilment of the Requirements

The Mobility Language will in the following be evaluated with regard to the requirements proposed in chapter 3 and summarised in appendix A.

Components: As in the distributed join-calculus, a component is modelled as a location, which has a code part (the definitions) and a state part (the processes). The Mobility Language has further made it possible to create separate code components (a location with an empty process part) and to reuse such code components in the creation of several other components.

The Mobility Language also allows one to express that a passive or executing component should be co-located with its code component in order to be executable. This is modelled by letting the passive component have a by-type reference to its code component. The Mobility Language thus fulfils all of the component requirements.

Locations: Locations organized in a hierarchic structure can be modelled the same way as in the distributed join-calculus. It is still not possible to model that interaction with certain components only is possible for components at the same location.

Mobility: The Mobility Language offers three migration primitives, which can be used to model the movement of all of a passive or executing component, a separate code component, or only the state of a component. Depending of the nature of the moved component, one can model movement of both stand-alone code and code segments. The possibility of specifying a continuation in the migration primitives also allows one to model that a code component or a passive component should start executing upon arrival to the destination location. Of the mobility mechanisms depicted in section 3.3, it is only the situation with weak mobility in figure 3.3 that can not be modelled directly in the Mobility Language.

Shipping and fetching of a component and synchronous moves can be modelled the same way as in the distributed join-calculus. The movement of a copy of a component can be modelled by first creating a copy of the component and then using a migration primitive to move the copy. Of the mobility requirements it is thus only the requirement that it should be possible to specify the destination as a set of locations fulfilling certain requirements, that isn't met.

Bindings: The marking of a component reference as by-type and the creation of component instances allow one to model by-type bindings. All components that are sub-locations of a moving component will also be moved and this can be used to model that certain referenced components should migrate together with the component. References to all other components will be kept across the network and it is thus not possible to express that certain references should become invalid. That a copy of a referenced component should be moved has to be modelled explicit; there is no mechanism in the language that allows it to happen implicitly when a component is moved.

Security: The security mechanisms are the same as for the distributed join-calculus.

Openness: Component and channel names can be created dynamically and be communicated between the components.

Communication: The addressing form used in the Mobility Language has been changed from the addressing form used in the distributed join-calculus. However, the communication is still one-to-one, and the addressing is location independent in the sense that the sender only has to know the name of the receiving component and not its location. The Mobility Language thus fulfils the same communication requirements as the distributed join-calculus.

Synchronization: The use of join-patterns to model mutual exclusion and condition synchronization is still possible.

Failure Handling: The failure handling part of the distributed join-calculus allows the modelling of both failure and failure detection.

Object Properties: Locations encapsulate data and operations and they thus provide a natural way of modelling objects. The new addressing form used to address a particular port in a particular location is similar to the addressing of a particular operation in a particular object. The possibility of having several locations that define the same port names will also allow polymorphism. The possibility of copying definitions from one location to another can be used to model inheritance.

The distributed join-calculus allows one to model that certain data or operations should be private simply by defining the corresponding ports in a local definition, i.e. by restricting the scope of the port names. This is still possible in the Mobility Language but it can not always be used in combination with by-type bindings from objects to the class definitions. If it should be possible to use the state of an object together with any copy of the class definition, it takes that all the copies of the class definition use the same port names. The class definition can thus not use a local definition to define ports in this case.

9.2 Possible Extensions to the Mobility Language

The Mobility Language fulfils more of the requirements than the distributed join-calculus do. The possibility of copying definitions and processes from one location to another has made it possible to reuse code components, to let one component inherit from another component, and to otherwise create copies of components. The introduction of by-type bindings and component instances has also made it possible to model that a code component and the state of a passive component move separately. There are, however, still some of the requirements that aren't fulfilled. The following sections will suggest a couple of extensions to the Mobility Language, which will allow further requirements to be fulfilled.

9.2.1 Restriction of Ports

As discussed in the last section in connection with the object requirements, it isn't always possible to model that a port should be private to a location. There are situations where one could be interested in letting a port name be known outside its defining location but where it anyway shouldn't be possible to send a message to the port from other locations. This is the case when code components are being instantiated or extended.

Consider the well-known object example:

$$\begin{array}{lll} \texttt{def} & \mathit{class}[& m_1\langle \mathit{args}\rangle | \mathit{state}\langle s\rangle \rhd \ldots \quad \wedge \\ & & m_2\langle \mathit{args}\rangle | \mathit{state}\langle s\rangle \rhd \ldots \quad : \mathsf{0}] \\ \wedge & \mathit{object}[\, \mathit{class} : & \mathit{state}\langle s_0\rangle] \\ \texttt{in} & \mathsf{0} & \end{array}$$

In this situation, the *state* port is known outside *class* and is used when the class is instantiated to an object with an initial state. However, even though the port name is known, it should not be possible to send messages to the *state* port from other locations. This suggests that a new language construct is needed.

The Mobility Language could be extended such that location definitions get the form:

$$a[D \setminus N : P : \sigma]$$

which will be written $\{\!|D|\!\} \setminus N \vdash^{\sigma}_{\psi a} \{\!|P|\!\}$ when the location is heated to solution form. N should be a set of port names defined in D. The intended meaning is that port names included in N are private to a, meaning that other locations cannot send messages to these ports. The port names are however known outside a. This should be reflected in the *Comm* rule by the addition of an extra side condition:

$$Comm: \quad \vdash_{\phi} a.x\langle\tilde{v}\rangle \parallel \setminus N \vdash_{\psi\epsilon a} \quad \rightarrow \quad \vdash_{\phi} \parallel \setminus N \vdash_{\psi\epsilon a} x\langle\tilde{v}\rangle \qquad (\phi \text{ alive}, x \notin N)$$

The extra side condition will prevent messages from being sent to a private port in another location.

The suggested restriction of port names is static in the sense that the set of private port names is determined once and for all when the location is defined, and the set can't be changed during computations. An extra heating rule should allow an empty set of private port names to be introduced into a location definition if none is given. This is similar to the introduction of an empty substitution in the *DefLoc* rule.

The set of private port names belongs to the definition part of the location, and the set should be copied too when the definitions are copied to another location. The *Copy* rule should therefore be changed to the following:

$$\begin{array}{l} \epsilon a[\,\beta \wedge D_1 \setminus N_1 : P_1 : \sigma_1] \vdash^{\sigma_3}_{\phi} \parallel c[D_2 \setminus N_2 : P_2 : \sigma_2] \vdash_{\psi} \qquad \rightharpoonup \\ \epsilon a[\,g_1(D_2 \wedge D_1\,,\, \sigma_2\sigma_{dl}) \setminus N_1 \cup N_2 : g_1(P_2|P_1\,,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \vdash^{\sigma_3}_{\phi} \\ \qquad\qquad\qquad\qquad \parallel c[D_2 \setminus N_2 : P_2 : \sigma_2] \vdash_{\psi} \\ (g_2(\beta, \sigma_3\sigma_1) = c,\ dom(\sigma_{dl}) = dl[D_2],\ range(\sigma_{dl}) \text{ fresh},\ \psi \text{ alive}) \end{array}$$

These extensions to the Mobility Language will allow the object in the previous example to be modelled such that other components can't send messages to the *state* port:

$$\begin{array}{ll} \texttt{def} & class[\ m_1\langle args\rangle|state\langle s\rangle \triangleright \ldots\ \wedge \\ & \qquad m_2\langle args\rangle|state\langle s\rangle \triangleright \ldots\ \setminus\{state\}\ : \mathtt{0}] \\ \wedge & object[\ class:\ state\langle s_0\rangle] \\ \texttt{in} & \mathtt{0} \end{array}$$

The extension will thus allow the Mobility Language to meet the requirement that data and operations can be made private to a component.

9.2.2 Restriction of Locations

The Mobility Language cannot model that certain components only should be accessible from components located at the same location. Also, it is not possible to express that a reference to a component should become invalid if the component moves or the component with the reference moves. This is really two sides of the same question. If it were possible to model that a component only is accessible from components at the same location, a reference to such a component would become invalid if either the component or the component with the reference moves to another location.

It is suggested that the Mobility Language is extended with a construct "$\gg b$" which can be used to restrict a location to only being accessible from within location b. Location definitions should thus have the form:

$$a[D \setminus N : P : \sigma] \gg b$$

which will be written $\{\!|D|\!\} \setminus N \vdash^{\sigma}_{\psi a} \gg b\ \{\!|P|\!\}$ when the location is heated to solution form. The intended meaning is that a is only accessible from processes in b or sublocations of b, i.e. messages cannot be sent to a from elsewhere and locations elsewhere cannot copy the definitions and processes from a. The location b should be an ancestor location of a. Formally this is stated as $\exists \phi_1, \phi_2 : \psi a = \phi_1 b \phi_2$ but in the following it will simply be written as $b \in \psi a$.

This restriction construct can be used to model that e.g. a printer only can be used from computers within its own network domain:

$$\begin{array}{l} domain_1[\ printer[print \rhd \ldots] \gg domain_1\ \wedge comp_1[\ \ldots\ ]\ \wedge \ldots\ ] \wedge \\ domain_2[\ comp_2[\ \ldots\ ]\ \wedge\ \ldots\ ] \end{array}$$

The *printer* in this example can be used by processes in $comp_1$ but not by processes in $comp_2$.

The intended meaning of the restriction construct should be reflected in the chemical rules of the MLCHAM. If a location is restricted, all its sublocations should also be restricted. This will be reflected when a location is heated to solution form:

$$DefSol: \quad \begin{array}{c} \epsilon a[D \setminus N : P : \sigma_2] \gg b_2\ \vdash^{\sigma_1}_{\phi} \gg b_1 \quad \rightharpoonup \\ \vdash^{\sigma_1}_{\phi} \gg b_1 \quad \| \quad \{\!|D|\!\} \setminus N \vdash^{\sigma_1\sigma_2}_{\phi\epsilon a} \gg max(b_1, b_2)\ \{\!|P|\!\} \\ (b_2 \in \phi,\ D \text{ is fully expanded}, \\ \text{no sublocation of } a \text{ is on the form } b[D_1 : P_1 | subst\langle c\rangle]) \end{array}$$

The function $max(\cdot,\cdot)$ should choose the one of the arguments that is the most restrictive location name. The extra side condition $b_2 \in \phi$ says that a location only can be restricted to a parent location. An extra heating rule should allow a restriction to the root location "$\gg$ " to be introduced in a location definition if no restriction is given.

The restriction of a location to another location should be reflected in the *Comm* rule by the addition of an extra side condition:

$$Comm: \quad \begin{array}{rcl} \vdash_{\phi} a.x\langle\tilde{v}\rangle & \| & \setminus N \vdash_{\psi\epsilon a} \gg b \quad \rightarrow \\ \vdash_{\phi} & \| & \setminus N \vdash_{\psi\epsilon a} \gg b\ x\langle\tilde{v}\rangle \end{array} \qquad (\phi \text{ alive},\ x \notin N,\ b \in \phi)$$

The extra side condition, $b \in \phi$, will prevent the delivery of a message that is sent from a location outside the "restriction sphere" of the destination location, i.e. only if the sender is a sublocation of b, will the message be delivered.

The restriction of a location to another location should also be reflected in the *Copy* rule. It should not be possible to copy the definitions from a location to somewhere outside the location's restriction sphere:

$$\begin{array}{l}
\epsilon a[\,\beta \wedge D_1 \setminus N_1 : P_1 : \sigma_1] \gg b_1 \vdash_{\phi}^{\sigma_3} \;\|\; c[D_2 \setminus N_2 : P_2 : \sigma_2] \gg b_2 \vdash_{\psi} \gg b_3 \quad \rightharpoonup \\
\epsilon a[\,g_1(D_2 \wedge D_1\,,\, \sigma_2\sigma_{dl}) \setminus N_1 \cup N_2 : g_1(P_2|P_1\,,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}] \gg b_1 \vdash_{\phi}^{\sigma_3} \\
\qquad\qquad\qquad\qquad\qquad\qquad \|\; c[D_2 \setminus N_2 : P_2 : \sigma_2] \gg b_2 \vdash_{\psi} \gg b_3 \\
(g_2(\beta, \sigma_3\sigma_1) = c,\; dom(\sigma_{dl}) = dl[D_2],\; range(\sigma_{dl}) \text{ fresh},\; \psi \text{ alive}, \\
max(b_2, b_3) \in \phi)
\end{array}$$

The extra side condition, $max(b_2, b_3) \in \phi$, assures that the definitions from c only can be copied to a location within c's restriction sphere. The restriction on c will not be transfered to the copy but if c has any sublocations that are restricted, these restrictions will be copied as they are.

The restriction of a location to another location should be allowed to change when the location moves. The `go` primitive should therefore be extended to take a third argument, which states the location to where the moving component should be restricted after the movement:

$$Move: \begin{array}{l}
a_1[\,c[D_2 \setminus N_2 : P_2|\mathtt{go}\langle a_2, b_4, \kappa\rangle : \sigma_2] \gg b_2 \;\wedge D_1 \setminus N_1 : P_1 : \sigma_1] \gg b_1 \;\vdash_{\phi}^{\sigma_4} \\
\| \quad \epsilon a_2[D_3 \setminus N_3 : P_3 : \sigma_3] \gg b_3 \;\vdash_{\psi} \quad \rightarrow \\
a_1[D_1 \setminus N_1 : P_1 : \sigma_1] \gg b_1 \;\vdash_{\phi}^{\sigma_4} \\
\| \quad \epsilon a_2[\,c[\,D_2 \setminus N_2 : P_2|\kappa\langle\rangle : \sigma_2] \gg b_4 \;\wedge D_3 \setminus N_3 : P_3 : \sigma_3] \gg b_3 \;\vdash_{\psi} \\
(D_2 \text{ is fully expanded, } c \notin range(\sigma_4\sigma_1)\text{, for all } b \in dl[D_2]\text{: } b \notin range(\sigma_4\sigma_1), \\
\phi \text{ alive, } b_4 \in \psi\epsilon a_2)
\end{array}$$

The extra side condition, $b_4 \in \psi\epsilon a_2$, says that c should still be restricted to a parent location after the movement.

These extensions to the mobility language will allow the modelling of the binding mechanism used in the Telescript technology. In this technology, an object can only be accessed from other objects and agents that are located at the same place as the object. When the owner agent of the object moves, the object will move too and all bindings to it from other agents will become invalid. In the following example, this is modelled as all agents being restricted to the place they occupy:

$$\begin{array}{l}
Telesphere[\\
\quad place_1[\; agent_1[\; obj[f\langle\rangle \rhd \ldots] \;\wedge \\
\qquad\qquad\qquad move\langle p, \kappa\rangle \rhd \mathtt{go}\langle p, p, \kappa\rangle \\
\qquad\qquad\quad : move(place_2); 0 \;] \gg place_1 \;\wedge \\
\qquad\quad agent_2[\; \ldots : obj.f\langle\rangle \;] \gg place_1 \\
\qquad] \qquad \wedge \\
\quad place_2[\; \ldots \;] \quad]
\end{array}$$

The agent $agent_1$ owns the object obj. Both the agent itself and the object can only be accessed from other agents or objects located at $place_1$. After $agent_1$ has moved, it will be restricted to $place_2$ and the message to obj from $agent_2$ can no longer be delivered.

9.2.3 References from a Location to Itself

In the examples in chapter 8 it has often been necessary for the copied definitions to be able to refer to the location to where they are copied. This has been the case in the remote cloning example in section 8.4 where the definition of *clone* in the *Agent* definition needs to know the name of the agent instance. In the example, it is not possible to use a by-type reference to *Agent* because agents created from the *Agent* definition shouldn't be seen as local agent instances because several agents can exist in the same location.

Instead, a change of the *Copy* rule can make it possible that a definition refer to its enclosing location also will do so after it has been copied to another location. An extra substitution should simply be applied to the copied definitions:

$$\mathit{Copy}: \quad \begin{array}{l} \epsilon a[\,\beta \wedge D_1 : P_1 : \sigma_1] \vdash^{\sigma_3}_{\phi} \parallel c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \quad \rightharpoonup \\ \epsilon a[\,g_1(D_2\{a/c\} \wedge D_1\,,\, \sigma_2\sigma_{dl}) : g_1(P_2\{a/c\}|P_1\,,\, \sigma_2\sigma_{dl}) : \sigma_1\sigma_2\sigma_{dl}]\, \vdash^{\sigma_3}_{\phi} \\ \qquad\qquad\qquad\qquad \parallel c[D_2 : P_2 : \sigma_2] \vdash_{\psi} \\ (g_2(\beta, \sigma_3\sigma_1) = c,\ \beta \neq `c,\ \mathit{dom}(\sigma_{dl}) = \mathit{dl}[D_2],\ \mathit{range}(\sigma_{dl}) \text{ fresh},\ \psi \text{ alive}) \end{array}$$

The substitution $\{a/c\}$ is applied to the copied definitions and processes. This will change any reference to the original enclosing location c to the new enclosing location a.

If this version of the *Copy* rule had been used with the agent example, any reference to *Agent* inside *Agent* would be changed to a reference to $agent_1$ when the definitions are copied to $agent_1$. The definition of the *clone* port would thus not have needed the first argument but could simply have used a reference to *Agent*:

$$\mathit{Agent}[\ \mathit{clone}\langle l, \kappa\rangle \rhd `\mathit{AgentRuntime.clone}\langle \mathit{Agent}, l, \kappa\rangle : \mathtt{0}\ ]$$

9.2.4 Branching

The Mobility Language can model infinite computations through the use of recursive or mutual recursive port definitions. It is also possible to model non-determinism by letting a location have several definitions of the same port. But there is no construct in the mobility language that allows branching.

Some form of conditional branching could be useful e.g. in order to determine when a recursive computation should finish. One possibility would be to have a construct that can check whether a certain message is present in the location. Such a construct could have the form `if` $m\langle x\rangle$ `then` P_1 `else` P_2, and the corresponding set of reaction rules would be something like the following:

$$\begin{array}{lll} \mathit{Ift}: & \vdash \{\!|(\texttt{if } m\langle x\rangle \texttt{ then } P_1 \texttt{ else } P_2)|P_3|\!\} \rightarrow \vdash \{\!|P_1|P_3|\!\} & (m\langle x\rangle \in P_3) \\ \mathit{Iff}: & \vdash \{\!|(\texttt{if } m\langle x\rangle \texttt{ then } P_1 \texttt{ else } P_2)|P_3|\!\} \rightarrow \vdash \{\!|P_2|P_3|\!\} & (m\langle x\rangle \notin P_3) \end{array}$$

Another possibility is to allow boolean values in the language and have a traditional construct like `if` b `then` P_1 `else` P_2.

9.3 Other Ways of Doing Things

Some choices have been made while constructing the language. The following will discuss a few of the things that could have been done differently in the Mobility Language.

9.3.1 Contract/Expand Primitives

The time where the *Copy* rule or the $Contract$ rule should be applied to a solution is governed by side conditions on some of the other chemical rules. The side condition "D is fully expanded" on the $DefSol$ rule dictates that all definitions should be copied to a location before any reductions can take place in the location. Likewise, the side conditions on the $Move$ rules will dictate that a location should be either expanded or contracted before the respective migration primitives can take effect.

Instead of letting the copying of definitions from one location to another work through heating/cooling rules and letting side conditions govern the time for the application of these rules, explicit `expand` and `contract` primitives could have been used. The semantics of the `expand` primitive would be given by a reaction rule that performs the same copying as is now given by the *Copy* rule and further sends a message on a given continuation channel. A reaction rule is irreversible and the corresponding contraction of a location would not be able to take place until a `contract` primitive were executed.

The use of explicit `expand` and `contract` primitives could be used to govern how much a location should be expanded or contracted before a migration. Only one migration primitive would be needed then, and the modeller could expand or contract the code as wanted before the movement. This could be very useful in certain situations.

Consider the situation where a copy of an object should be moved. This cannot be done simply with the following definition:

$$obj_2[obj : \mathtt{gostate}(jvm_2); \mathsf{0}]$$

The result of this definition would be that obj_2 moves as it is, without the definitions and processes from *obj* having been copied to obj_2. The intended form of obj_2 can only be achieved by first expanding obj_2 and then contracting it again. To obtain the intended result in the Mobility language one would thus have to let the `gostate` primitive appear as result of a reduction in obj_2. If explicit `expand` and `contract` primitives had been used in the Mobility Language, the intended movement could have been expressed with the following:

$$obj_2[obj : \mathtt{expand;\ contract;\ gostate}(jvm_2); \mathsf{0}]$$

This approach with explicit `expand` and `contract` primitives seems to have certain advantages, but it also has some drawbacks. It would now be up to the modeller to ensure that a location is fully expanded before any reductions take place in the location. This takes much more work by the modeller because it would be necessary to keep track on whether a location is fully expanded, and a model would grow because of the extra `expand` and `contract` primitives.

9.3.2 The Style of the Semantics

The semantics of the Mobility Language has been described in terms of a chemical abstract machine. This has been chosen because this is the approach used for the distributed join-calulus that was taken as the basis for the Mobility Language. Also, the chemical abstract machine can very naturally express the parallelism of the processes within a location and of the locations themselves.

However, The chemical rules for the MLCHAM have grown rather big and complicated and a lot of extra heating/cooling rules had to be added. This was partly because one of the main advantages of the CHAM, the natural associativity and commutativity, has disappeared because of the need to rearrange the terms of a location while the location is still in definition form.

One could therefore imagine that other styles of semantics could be just as appropriate as the given chemical semantics for the Mobility Language. A more classical structural operational semantics similar in style to the semantics given for the actor formalisms could perhaps be used. However, this would require a lot of extra work and for now, the MLCHAM works fine.

9.4 Future Work

The Mobility Language can be used as it is to model distributed systems with mobile code. However, the use of the language could benefit from further work in certain areas.

9.4.1 Validation of a Model

The possible behaviours of a system modelled in the Mobility Language are given by the formal semantics of the language. The formal semantics allows the building of simulation tools that can simulate a system and show some of the possible computations paths (i.e. the development of the chemical solution representing the model). A simulation tool can help in the validation of a model but it cannot formally prove that a model has the desired behaviour.

In order to formally verify a model, the intended behaviour of the system needs a formal representation, and some proof rules are needed in order to determine whether the model satisfies the requirements. The desirable behaviour of a system can be represented by another model written in the Mobility Language. This model should be simple and should express nothing but the intended behaviour. An equivalence relation between processes could then be used to determine whether the model of the system and the requirement model behave the same.

This approach builds on the existence of an equivalence relation between processes. Two processes should be equal when they behave the same. The formalisms described in chapter 4 all have a definition of observation equivalence which relates processes with the same observable behaviour. The distributed join-calculus also defines an observation equivalence, and this equivalence relation could be modified to also cover processes in the Mobility Language. The observation equivalence would have to be changed a little because the addressing of messages has been changed. A process could now be characterised by its ability to emit a message to a certain port in a certain location.

The definition of a proper equivalence between processes in the Mobility Language would allow reasoning about whether a model has the intended behaviour. This could for example be used to prove that the RMI mechanisms work correctly, i.e. that the invocation of an operation in a stub object gives the same result as a direct invocation of the operation in the server object.

9.4.2 Type System

The Mobility Language operates with three kinds of values: port names, location names, and combinations. These kinds of values cannot be used interchangeable because only port names and combinations can be used as the addressing part of a message, and only location names can be used to indicate copying. The language primitives also need arguments of the correct kind.

A modeller should take care that the correct kinds of values are used in the correct places. The provision of a type system for the Mobility Language can ease this task. When a message is sent to a port, the values contained in the message should be of the correct kind. Instead of having to look through the port definition in order to decide the needed kinds of values, the value types could be indicated in the join-pattern where the port is defined.

An extension of the language with a type system will help ensure that a model is semantically correct. The type check can also be automated using some tool, which at the same time can check that the model satisfies the well-formed conditions of the Mobility Language.

Chapter 10

Conclusion

This project has been concerned with the construction of a modelling language, which should have the ability to describe systems with code and process mobility.

The construction of the language has been based on a set of requirements which has been identified through a study of existing technologies with mobility. The requirements have concerned which concepts it should be possible to model; these concepts include components of a certain structure, mobility of components or parts of components, and different kinds of bindings between components.

Existing formal languages that can describe mobility have also been studied in order to identify the language constructs that allow the different mobility mechanisms to be modelled. This process has resulted in the choosing of the distributed join-calculus as the basis of the new modelling language because it already fulfils a lot of the posed requirements.

The distributed join-calculus has been extended and modified in order to meet more of the language requirements. The result has been the Mobility Language. The main achievement of the Mobility Language is that it adds the following abilities to the distributed join-calculus:

- It has become possible to copy definitions and processes from one location to another, thereby allowing code reuse, inheritance, and copying of components.
- Modelling of by-type bindings has become possible through the introduction of component instances and by-type markings of component references.
- Movement of the different parts of a component has become possible through the introduction of three different migration primitives and through the use of by-type bindings and copying.

The semantics of the Mobility Language has been given in terms of a chemical abstract machine, the MLCHAM.

The Mobility Language does not meet all of the posed requirements but the main objective has been achieved: The Mobility Language can describe the mobility mechanisms used in Java. Extensions to the Mobility Language have also been suggested in order to allow more of the requirements to be fulfilled. Future work regarding the Mobility Language includes the building of tools that can simulate a model or verify that a model has the desired behaviour.

The construction of the Mobility Language has been a very interesting process, which has touched upon many areas of computer science.

Bibliography

[Agh99] G. A. Agha. Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45(15):1263–1277, 1999.

[AMST97] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley Longman, Inc., 2000.

[ATZ01] G. A. Agha, P. Thati, and R. Ziaei. Actors: A model for reasoning about open distributed systems. In *Formal Methods for Distributed Processing* [BD01]. 2001.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–48, 1992. Available at http://www.inria.fr/meije/personnel/Gerard.Berry/cham.ps.

[BCF02] Nick Benton, Luca Cardelli, and Cédric Founet. Modern concurrency abstractions for C$^\sharp$. In B. Magnusson, editor, *ECOOP 2002 - Object Oriented Programming. 16th European Conference. Proceedings (Lecture Notes in Computer Science Vol.2374)*, pages 415–40. Springer-Verlag, 2002.

[BD01] H. Bowman and J. Derrick, editors. *Formal methods for distributed processing : A survey of object-oriented approaches*. Cambridge University Press, 2001.

[Cal99] California Software Co. Ltd. *UPnP, Jini and Salutation - White Paper*, June 1999. http://www.calsoft.co.in/techcenter/Ebiz/upnp.html.

[CG01] L. Cardelli and A. D. Gordon. Mobile ambients. In *Formal Methods for Distributed Processing* [BD01]. 2001.

[Emm00] W. Emmerich. *Engineering distributed objects*. Wiley, 2000.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, January 21-24 1996.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer-

Verlag, August 26-29 1996. LNCS 1119.

[FH00] Eric Freeman and Susanne Hupfer. *Make room for JavaSpaces, Part 4: Explore Jini transactions with JavaSpaces.* JavaWorld, April 2000. http://www.javaworld.com/javaworld/jw-04-2000/jw-0421-jiniology.html.

[FM97] Cédric Fournet and Luc Maranget. *The Join-Calculus language release 1.05, Documentation and user's manual.* INRIA, 1997. Available at http://pauillac.inria.fr/join/manual/index.html.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–61, 1998.

[Gen95] General Magic, Inc. *The Telescript Language Reference Version 1.0*, October 1995. http://www.science.gmu.edu/m̃chacko/Telescript/docs/-telescript.html.

[GKCR00] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. Mobile Agents: Motivations and State-of-the-Art Systems. Technical Report TR2000-365, Dartmouth College, Computer Science, April 2000. ftp://ftp.cs.dartmouth.edu/TR/TR2000-365.ps.Z.

[GZ01] M. Gaspari and G. Zavattaro. An actor algebra for specifying distributed systems: The hurried philosophers case study. In G. A. Agha, F. De Cindio, and G. Rozenberg, editors, *Concurrent object-oriented programming and petri nots. Advances in petri nets (Lecture Notes in Computer Science Vol.2001)*, pages 428–44. Springer-Verlag, 2001.

[Kim97] Wooyoung Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, 1997. http://www.cs.uiuc.edu/Dienst/Repository/-2.0/Body/ncstrl.uiuc_cs/UIUCDCS-R-97-2001/postscript.ps.

[NM00] Ulf Nilsson and Jan Maluszynski. *Logic Programming and Prolog.* 2000. http://www.ida.liu.se/˜ulfni/lpp/.

[Pic01] Gian Pietro Picco. Mobile agents: an introduction. *Microprocessors and Microsystems*, 25(2):65–74, 2001.

[Rad99] Matthias Radestock. *Coordination in Adaptive Open Distributed Systems*, 1999. The section "Introduction To Rosette" is available at http://www.lshift.net/docs/mr/Main/node147.html.

[San01] Davide Sangiorgi. Asynchronous process calculi: the first- and higher-order paradigms. *Theoretical Computer Science*, 253(2):311–350, 2001.

[Sew01] Peter Sewell. π-calculi. In *Formal methods for distributed processing [BD01].* 2001.

[Som01] Frank Sommers. *Object mobility in the Jini environment.* JavaWorld, January 2001. http://www.javaworld.com/javaworld/jw-01-2001/jw-0105-jiniology.html.

[Sun] Sun Microsystems. *Dynamic code downloading using RMI.* http://java.sun.com/j2se/1.3/docs/guide/rmi/codebase.html.

[Sun99] Sun Microsystems. *Java™Remote Method Invocation Specification*, December 1999. http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmititle.html.

[Sun01a] Sun Microsystems. *Jini™Architecture Specification Version 1.2*, December 2001. http://wwws.sun.com/software/jini/specs/jini1.2html/-jini-title.html.

[Sun01b] Sun Microsystems. *Jini™Technology Core Platform Specification version 1.2*, December 2001. http://wwws.sun.com/software/jini/specs/-jini1.2html/core-title.html.

[Sun02] Sun Microsystems. *JavaSpacesTM Service Specification Version 1.2.1*, April 2002. http://wwws.sun.com/software/jini/specs/jini1.2html/js-title.html.

[SWP99] Peter Sewell, Pawel T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Lecture Notes in Computer Science*, volume 1686, pages 1–31. 1999.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf.

[Ven97] Bill Venners. *The architecture of aglets.* JavaWorld, April 1997. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html.

[Wan99] Paul S. Wang. *Java with Object-Oriented Programming and World Wide Web Applications.* PWS Publishing, 1999.

[Whi97] J. E. White. Mobile agents. In J. M. Bradshaw, editor, *Software Agents.* AAAI Press / The MIT Press, 1997.

[Woj00] Pawel T. Wojciechowski. *The Nomadic Pict System Release 1.0-alpha, Documentation and user's manual*, December 2000. Available at http://lsrwww.epfl.ch/~pawel/npict-tutorial.ps.

Appendix A

Requirements to a Modelling Language

The following is a summary of all the requirements identified in chapter 3. The requirements are grouped into categories, which will be used in the analysis of existing formal languages (chapter 4).

Components
It should be possible to model passive components consisting of a code component and a state.
It should be possible to model executing components consisting of a passive component and control information.
It should be possible to combine several components into a concurrently executing system.
It should be possible to express dynamic creation of passive and executing components.
It should be possible to use the same code component in several passive components.
The language should reflect that a passive component can only be executed if both the code component and the state is present at the same location.

Locations
The language should have a way of modelling that some components and resources are co-located while others are at different locations.

It should be possible to express a hierarchic structure in the locations.
It should be possible to specify that interaction with certain components or resources only is possible from within the same location.

Mobility
It should be possible to model that a component or part of a component moves from one location to another location.
It should be possible to model the movement of the following parts: a code component, the state of a passive component, all of a passive component and all of an executing component.
It should be possible to model whether the moved part is a code segment that will become part of a passive component at the destination or whether it is stand-alone code that will become part of a new executing component.
Components should have a way of naming which location itself or other components should move to. It should be possible to name both a particular location and a set of locations that fulfil some requirements.
It should be possible to specify that a component migrates at its own will, that it is being shipped by some other executing component, and that it is being fetched by some other executing component.
It should be possible to model that a copy of a component (passive or executing) is being moved.
It should be possible to model both synchronous and asynchronous moves.

Bindings
It should be possible to model by-type bindings.
It should be possible to model that a referenced resource or component migrates together with a component.
It should be possible to model that a copy of a referenced resource or component migrates together with a component.
It should be possible to model that a component keeps a reference across the network to a resource or a component, which doesn't migrate together with the component.

It should be possible to model that a reference to a resource or a component, which doesn't migrate together with the component, becomes invalid when the component moves.

Security
It should be possible to deny an executing component access to a location.
It should be possible to limit a components capabilities regarding access to resources and creation of new executing components.

Openness
The language should allow dynamic creation of new names.
The language should allow names (or what else is necessary for establishing new communication links) to be communicated between components.

Communication
The language should provide a way of addressing whom to communicate with, allowing both one-to-one, one-to-many, and one-to-one-of-many communication.
It should be possible to model both one-way communication and two-way communication where an operation is invoked and a reply returned.
It should be possible to model both synchronous and asynchronous communication.
The language should support both location dependent as well as location independent addressing.

Synchronization
It should be possible to model mutual exclusion.
It should be possible to model condition synchronization.

Failure Handling

It should be possible to model failure in a component.
The language should provide a way of letting other components become aware of failure in a component.

Object Properties
It should be possible to model encapsulation of data and operations working on the data.
It should be possible to model that some data or operations are private to a component, i.e. can't be accessed from other components.
A component should be able to inherit code from other components.
The language should allow polymorphism.

Appendix B

Comparison of the Formalisms

REQUIREMENT	*π-calculus*	*Nomadic π*	*Distributed join-calculus*	*Actors*[a)]	*Ambients*
Components					
Can model passive components.	+	+	+	+	+
Can model executing components.	+	+	+	+	+
Allows concurrent composition of components.	+	+	+	+	+
Allows dynamic creation of components.	+	+	+	+	+
Allows a code component to be reused in several components.	+[b)]	+	-	+	-
Reflects that code and state should be co-located in order to execute.	-	-	+	+	-
Locations					
Can model that components are at different locations.	+[c)]	+	+	+[c)]	+
Allows a hierarchic structure of the locations.	-	-	+	-	+
Can express that interaction with certain components only is possible from within the same location.	+[c)]-	-	-	-	-
Mobility					

Requirement	π-calculus	Nomadic π	Distributed join-calculus	Actors[a)]	Ambients
Can model the movement of a passive or executing component.	+[c)]	+	+	+[c)]	+
Can model the movement of a code component.	+[d)]	-	-	-	-
Can model the movement of the state of a component.	-	-	-	-	-
Can model movement of both stand-alone code and code segments.	+[c)]	-	+	+[c)]	+
Can specify the destination as a particular location.	+[c)]	+	+	+[c)]	+
Can specify the destination as a set of locations fulfilling some requirements.	-	-	-	-	-
Can model shipping and fetching of a component.	+[c)]	+	+	+[c)]	+
Can model that a component moves at its own will.	+[c)]	+	+	+[c)]	+
Can model that a copy of a component is being moved.	+[d)]	-	-	+	-
Can model asynchronous moves.	+[c)]	+	+	+[c)]	+
Can model synchronous moves.	+[c)]	+	+	+[c)]	+
Bindings					
Can model by-type bindings.	-	-	-	-	+
Can model that a referenced component migrates together with the component.	-	-	+	-	+
Can model that a copy of a referenced component migrates together with the component.	-	-	-	-	-
Can model that a binding is kept across the network when a component moves.	+[c)]	+	+	+[c)]	+
Can model that a binding becomes invalid when a component moves.	-	-	-	-	+
Security					
Can deny a component access to a location.	+[c)]	-	+	+[c)]	+
Can limit a components capabilities.	-	-	-	-	+

Requirement	π-calculus	Nomadic π	Distributed join-calculus	Actors[a)]	Ambients
Openness					
Allows dynamic creation of new names.	+	+	+	+	+
Allows communication of names (or other contact information) between components.	+	+	+	+	+
Communication					
Supports one-to-one communication.	+	+	+	+	+
Supports one-to-one-of-many communication.	+	-	-	-	+
Supports one-to-many communication.	-	-	-	-	-
Supports one-way communication.	+	+	+	+	+
Supports two-way communication, particularly the invocation of an operation and the return of a result.	-[e)]	-[e)]	+	-[e)]	-[e)]
Supports synchronous communication.	+	-	-	-	-
Supports asynchronous communication.	+	+	+	+	+
Allows location dependent addressing.	-	+	-	-	+
Allows location independent addressing.	+	+	+	+	-
Synchronization					
Allows modelling of mutual exclusion.	+	+	+	+	+
Allows modelling of condition synchronization.	+	+	+	-	+
Failure Handling					
Allows modelling of failure in a component.	+	+	+	+	-
Allows a component to detect failure in other components.	-	-	+	-	-
Object Properties					
Can model encapsulation of data together with operations working on the data.	+	+	+	+	+
Can model that some data and operations are private to a component.	+	+	+	-	+
Supports inheritance.	+	-	-	-	-
Allows polymorphism.	-	-	-	+	+

[a]: The basic actor model with no extensions.
[b]: Only polyadic π-calculus.
[c]: The language doesn't have an explicit location concept or migration primitives, but it can be modelled.
[d]: Only higher order π-calculus.
[e]: Two-way communication can be modelled using a continuation passing style, but no syntax directly supports this.

Appendix C

The Distributed Join-Calculus

The syntax and semantics of the distributed join-calculus[FGL+96] is presented here in its entirety.

Syntax

The existence of the following set of names will be assumed:

- $\mathcal{N}$, an infinite set of port names ranged over by the variables $x, y, \ldots$ If the port is used as a continuation it wil be denoted by κ. A port has an arity telling the number of values contained in a message to that port.
- $\mathcal{L}$, an infinite set of location names ranged over by the variables $a, b, \ldots$ A marker $\Omega \notin \mathcal{L}$ is used to tag failed locations. For every $a \in \mathcal{L}$, ϵa denotes either a or Ωa. Finite strings of such possibly tagged location names are denoted by $\phi, \psi, \ldots$

Values in the distributed join-calculus are port names and location names. $u, v, \ldots$ will range over such values and $\tilde{v}$ will denote a tuple of such values.

The terms of the distributed join-calculus are processes P, containing definitions D and join patterns J. They are defined by the following abstract syntax:

$P ::=$	$P \mid P$	composition	$D ::=$	$D \wedge D$	co-definition
$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	definition	$\mid$	$J \triangleright P$	local rule
$\mid$	$x\langle\tilde{v}\rangle$	message	$\mid$	$\texttt{T}$	inert definition
$\mid$	$\texttt{0}$	inert process	$\mid$	$a[D : P]$	sub-location
$\mid$	$\texttt{go}\langle a, \kappa\rangle$	migration	$\mid$	$\Omega a[D : P]$	dead sub-location
$\mid$	$\texttt{halt}\langle\rangle$	termination	$J ::=$	$J \mid J$	join-pattern
$\mid$	$\texttt{fail}\langle a, \kappa\rangle$	failure detection	$\mid$	$x\langle\tilde{v}\rangle$	message pattern

Scoping of variables

The below definitions define what is ment by the free variables $fv[.]$ in a process or a definition, the defined variables $dv[.]$ in a definition or a join pattern, and the received variables $rv[.]$ in a join pattern. These definitions will be used to define the scope of variables.

In the definition, $S_1 \cup S_2$ will denote union of any two sets, whereas $S_1 \uplus S_2$ will denote the union of two disjoint sets.

$$
\begin{array}{lllll}
J: & dv[J_1|J_2] \stackrel{\text{def}}{=} dv[J_1] \cup dv[J_2] & & rv[J_1|J_2] \stackrel{\text{def}}{=} rv[J_1] \uplus rv[J_2] \\
& dv[x\langle\tilde{v}\rangle] \stackrel{\text{def}}{=} \{x\} & & rv[x\langle\tilde{v}\rangle] \stackrel{\text{def}}{=} \{u \in \tilde{v}\} \\
D: & dv[D_1 \wedge D_2] \stackrel{\text{def}}{=} dv[D_1] \cup dv[D_2] & & fv[D_1 \wedge D_2] \stackrel{\text{def}}{=} fv[D_1] \cup fv[D_2] \\
& dv[J \triangleright P] \stackrel{\text{def}}{=} dv[J] & & fv[J \triangleright P] \stackrel{\text{def}}{=} dv[J] \cup (fv[P] - rv[J]) \\
& dv[\mathtt{T}] \stackrel{\text{def}}{=} \emptyset & & fv[\mathtt{T}] \stackrel{\text{def}}{=} \emptyset \\
& dv[a[D:P]] \stackrel{\text{def}}{=} \{a\} \uplus dv[D] & & fv[a[D:P]] \stackrel{\text{def}}{=} \{a\} \cup fv[D] \cup fv[P] \\
P: & fv[x\langle\tilde{v}\rangle] \stackrel{\text{def}}{=} \{x\} \cup \{u \in \tilde{v}\} & & fv[\mathtt{0}] \stackrel{\text{def}}{=} \emptyset \\
& fv[P_1|P_2] \stackrel{\text{def}}{=} fv[P_1] \cup fv[P_2] & & fv[\mathtt{def}\ D\ \mathtt{in}\ P] \stackrel{\text{def}}{=} (fv[P] \cup fv[D]) - dv[D] \\
& fv[\mathtt{go}\langle a, \kappa\rangle] \stackrel{\text{def}}{=} \{a, \kappa\} & & fv[\mathtt{fail}\langle a, \kappa\rangle] \stackrel{\text{def}}{=} \{a, \kappa\} \\
& fv[\mathtt{halt}\langle\rangle] \stackrel{\text{def}}{=} \emptyset & &
\end{array}
$$

The use of $\uplus$ in the definition of $rv[J_1|J_2]$ means that a join pattern can't be used to match the contents of messages, e.g. the join pattern $x\langle v\rangle \mid y\langle v\rangle$ is not allowed. Likewise, the definition of $dv[a[D:P]]$ requires that a location variable can't be redefined inside the location.

Besides the disjointness requirements given by the above definitions, there are some extra conditions that must be fulfilled by a well-formed definition:

- Location variables can be defined only once in a scope, i.e. $\mathtt{def}\ a[D_1 : P_1] \wedge a[D_2 : P_2]\ \mathtt{in}\ P$ isn't allowed.
- Port variables can only appear in the join-patterns of one location, i.e. it is allowed to write $\mathtt{def}\ x\langle\rangle \triangleright P_1 \wedge x\langle\rangle|y\langle\rangle \triangleright P_2\ \mathtt{in}\ P$, but $\mathtt{def}\ x\langle\rangle \triangleright P_1 \wedge a[x\langle\rangle|y\langle\rangle \triangleright P_2 : P_3]\ \mathtt{in}\ P$ isn't allowed.

Semantics

The operational semantics of the distributed join-calculus is given by a chemical abstract machine, the distributed reflexive CHAM (DRCHAM). The terms of the distributed join-calculus constitute the molecules of the DRCHAM.

The state of the DRCHAM is represented by a multiset of labeled solutions, where each labeled solution represents a location in the distributed join-calculus. A labeled solution has the form $\mathcal{D} \vdash_\phi \mathcal{P}$, where $\mathcal{D}$ is a multiset of active reduction rules and $\mathcal{P}$ is a multiset of running processes. The string of location names, ϕ, holds the names of the parent locations ending with the name of the current location. A location $\vdash_\phi$ is a sublocation of another location $\vdash_\psi$ if ψ is a prefix of ϕ. The labeled solutions in a DRCHAM are separated by $\|$.

The heating/cooling rules ($\rightleftharpoons$) and reaction rules ($\rightarrow$) of the DRCHAM are listed below. The heating/cooling rules express a structural equivalence between terms in the solutions; the reaction rules give the basic computation steps.

The rules are presented in a style where only the parts of the chemical solution that are affected by the rule will be shown, i.e. an implicit context is assumed. E.g. the rule $\mathcal{D} \vdash \mathcal{P} \cup \{\!|P_1|P_2|\!\} \rightleftharpoons \mathcal{D} \vdash \mathcal{P} \cup \{\!|P_1, P_2|\!\}$ will be written $\vdash P_1|P_2 \rightleftharpoons \vdash P_1, P_2$.

`str-join`	$\vdash P_1 \| P_2$	$\rightleftharpoons$	$\vdash P_1, P_2$	
`str-null`	$\vdash \texttt{0}$	$\rightleftharpoons$	$\vdash$	
`str-and`	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$	
`str-nodef`	$\texttt{T} \vdash$	$\rightleftharpoons$	$\vdash$	
`str-def`	$\vdash \texttt{def}\ D\ \texttt{in}\ P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$	($range(\sigma_{dv})$ fresh)
`str-loc`	$\epsilon a[D : P] \vdash_\phi$	$\rightleftharpoons$	$\vdash_\phi \ \| \ \{\!\|D\|\!\} \vdash_{\phi\epsilon a} \{\!\|P\|\!\}$	(a frozen)
`red`	$J \rhd P \vdash J\sigma_{rv}$	$\rightarrow$	$J \rhd P \vdash P\sigma_{rv}$	(ϕ alive)
`comm`	$\vdash_\phi x\langle\tilde{v}\rangle \ \| \ J \rhd P \vdash$	$\rightarrow$	$\vdash_\phi \ \| \ J \rhd P \vdash x\langle\tilde{v}\rangle$	($x \in dv[J]$, ϕ alive)
`move`	$a[D : P\|\texttt{go}\langle b, \kappa\rangle] \vdash_\phi \ \| \ \vdash_{\psi\epsilon b}$	$\rightarrow$	$\vdash_\phi \ \| \ a[D : P\|\kappa\langle\rangle] \vdash_{\psi\epsilon b}$	(ϕ alive)
`halt`	$a[D : P\|\texttt{halt}\langle\rangle] \vdash_\phi$	$\rightarrow$	$\Omega a[D : P] \vdash_\phi$	(ϕ alive)
`detect`	$\vdash_\phi \texttt{fail}\langle a, \kappa\rangle \ \| \ \vdash_{\psi\epsilon a}$	$\rightarrow$	$\vdash_\phi \kappa\langle\rangle \ \| \ \vdash_{\psi\epsilon a}$	($\psi\epsilon a$ dead, ϕ alive)

In the side condition "$range(\sigma_{dv})$ fresh", σ_{dv} is a substitution that instantiates the port variables and location variables $dv[D]$ to distinct, fresh names. By "fresh" is ment a name that isn't a free variable in `def` D `in` P or in any of the processes or definitions in the context. The substitution σ_{rv} in the rule `red` substitutes the actual transmitted names for the received variables $rv[J]$.

The side condition "a frozen" means that there are no solutions of the form $\vdash_{\phi a \psi}$ where ψ is a non-empty label.
The side condition "ϕ dead" means that the string ϕ contains the marker Ω; "ϕ alive" means that ϕ doesn't contain the marker Ω.

Appendix D

Example Computations

D.1 Ex.1 Object Mobility in Java

The following shows the necessary steps when performing the computations for the example in section 8.1. The starting point is the chemical solution $\{\!|\ \ |\!\} \vdash^{\{\}} \{\!|P|\!\}$ where P is the process from the example. The solution will be heated and cooled as necessary in order to make the molecules react. The listing below describes each step in words; the development of the chemical solution can be seen on the following pages. The heating/cooling rules $Join$, $Null$, $DefAnd$, $DComm$, $DAssoc$, $DefNull$, and $DefNull2$ will be applied to the solution whenever needed; all other rules will be mentioned explicitly when they are applied.

1. The solution is heated according to the rules Def and $DefLoc$. This introduces empty substitutions into each location definition and introduces new names for all the defined variables. The applied substitution is:
$$\begin{aligned}\sigma_{dv} = \ & \{c_{12}/c_1,\ c_{22}/c_2,\ c_{32}/c_3,\ library_2/library,\ jvm_{12}/jvm_1,\\ & jvm_{22}/jvm_2,\ obj_2/obj,\ move_2/move,\ f_2/f,\ state_2/state\}\end{aligned}$$
2. The solution is heated according to the $Copy$ rule. The definitions from $library$ are copied to jvm_1 and jvm_2 and new names are introduced for the copied locations. The connection between original location names and the names of the copies are stored in the substitutions:
$$\begin{aligned}\sigma_1 = \ & \{c_{13}/c_{12},\ c_{23}/c_{22},\ c_{33}/c_{32}\} \text{ and}\\ \sigma_2 = \ & \{c_{14}/c_{12},\ c_{24}/c_{22},\ c_{34}/c_{32}\}\end{aligned}$$
The by-type reference in obj_2 to c_{22} will not be changed by the substitutions.
3. The location definition of jvm_{12} is heated to solution form using the rules $DefSol$ and $DefLoc$. This will make jvm_{12} ready to allow computations to be performed.
4. The solution $\vdash_{jvm_{12}}$ is heated according to the $Copy$ rule. $g(\grave{}c_{22}, \sigma_1) = c_{23}$, so the definitions from c_{23} are copied to obj_2. There are no sublocations that need to be given new names.

5. The location definition of obj_2 is heated to solution form using the $DefSol$ rule. The solution will inherit the substitution σ_1 from its parent location.
6. The message to obj_2 can now finally be sent using the $Comm$ rule.
7. A reduction is performed in the solution $\vdash_{jvm_{12}obj_2}$ according to the Red rule and the substitution $\sigma_{rv} = \{jvm_{22}/l\}$. The guarded process does not contain any by-type references so application of the substitution σ_1 to the process will not have any effect.
8. The solution $\vdash_{jvm_{12}obj_2}$ is cooled according to the $SolDef$ rule.
9. The location definition obj_2 is cooled according to the $Contract$ rule. The other involved location definition (the location c in the $Copy$ rule) could be c_{22}, c_{23} or c_{24} (if $library$ and jvm_{22} where heated to solution form). The side condition $b \notin range(\sigma_1)$ removes the possibility of b being c_{23}, and the side condition $b \notin range(\sigma_4)$ removes the possibility of $c = b$ being c_{24}. There are no substitutions that need to be "reapplied".
10. The solution $\vdash_{jvm_{12}}$ is cooled using the $SolDef$ rule.
11. The $MoveS$ rule is applied and obj_2 moves from jvm_{12} to jvm_{22} bringing only its state.
12. The location definition jvm_{22} is heated to solution form using the $DefSol$ and $DefLoc$ rules.
13. The solution $\vdash_{jvm_{22}}$ is heated according to the $Copy$ rule. $g(`c_{22}, \sigma_2) = c_{24}$, so the definitions from c_{24} are copied to obj_2. There are no sublocations that need to be given new names.
14. The location definition obj_2 is heated to solution form (using the rule $DefSol$) to allow computations to take place in obj_2.
15. Finally a reduction on the port f_2 in obj_2 can take place.

$$
\begin{aligned}
&\{\!|\ \ |\!\} \vdash^{\{\}} \{\!|\ \mathtt{def}\ library\ [\ c_1[D_1 : \mathsf{0}]\ \wedge\ c_2[move\langle l\rangle \triangleright \mathtt{gostate}\langle l, f\rangle \wedge f\langle\rangle | state\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_3[D_3 : \mathsf{0}] : \mathsf{0}\ ] \\
&\qquad\qquad \wedge\ jvm_1\ [\ library \wedge obj[\grave{}\, c_2 : state\langle s_0\rangle] : \mathsf{0}\ ] \\
&\qquad\qquad \wedge\ jvm_2\ [\ library : \mathsf{0}\ ] \\
&\qquad\qquad \mathtt{in}\ obj.move\langle jvm_2\rangle\ |\!\}
\end{aligned}
$$

$\xrightarrow{1}{}^{*}$ *Def*

$$
\begin{aligned}
&\{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ], \\
&\quad jvm_{12}\ [\ library_2 \wedge obj_2[\grave{}\, c_{22} : state_2\langle s_0\rangle] : \mathsf{0} : \{\}\ ], \\
&\quad jvm_{22}\ [\ library_2 : \mathsf{0} : \{\}\ ]\ |\!\} \vdash^{\{\}} \{\!|\ obj_2.move_2\langle jvm_{22}\rangle\ |\!\}
\end{aligned}
$$

$\xrightarrow{2}{}^{*}$ *Copy*

$$
\begin{aligned}
&\{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ], \\
&\quad jvm_{12}\ [\ c_{13}[D_1 : \mathsf{0}]\ \wedge\ c_{23}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{33}[D_3 : \mathsf{0}] \wedge \\
&\qquad\qquad obj_2[\grave{}\, c_{22} : state_2\langle s_0\rangle] : \mathsf{0} : \sigma_1\ ], \\
&\quad jvm_{22}\ [\ c_{14}[D_1 : \mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{34}[D_3 : \mathsf{0}] : \mathsf{0} : \sigma_2;] \\
&\quad |\!\} \vdash^{\{\}} \{\!|\ obj_2.move_2\langle jvm_{22}\rangle\ |\!\}
\end{aligned}
$$

$\xrightarrow{3}{}^{*}$ *DefSol*

$$
\begin{aligned}
&\quad \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ], \\
&\qquad jvm_{22}\ [\ c_{14}[D_1 : \mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{34}[D_3 : \mathsf{0}] : \mathsf{0} : \sigma_2\ ] \\
&\qquad |\!\} \vdash^{\{\}} \{\!|\ obj_2.move_2\langle jvm_{22}\rangle\ |\!\} \\
&\| \ \{\!|\ \ c_{13}[D_1 : \mathsf{0} : \{\}],\ c_{23}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}],\ c_{33}[D_3 : \mathsf{0} : \{\}], \\
&\qquad obj_2[\grave{}\, c_{22} : state_2\langle s_0\rangle : \{\}]\ |\!\} \vdash^{\sigma_1}_{jvm_{12}} \{\!|\ \ |\!\}
\end{aligned}
$$

$\xrightarrow{4}$ *Copy*

$$
\begin{aligned}
&\quad \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ], \\
&\qquad jvm_{22}\ [\ c_{14}[D_1 : \mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}]\ \wedge\ c_{34}[D_3 : \mathsf{0}] : \mathsf{0} : \sigma_2\ ] \\
&\qquad |\!\} \vdash^{\{\}} \{\!|\ obj_2.move_2\langle jvm_{22}\rangle\ |\!\} \\
&\| \ \{\!|\ \ c_{13}[D_1 : \mathsf{0} : \{\}],\ c_{23}[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}],\ c_{33}[D_3 : \mathsf{0} : \{\}], \\
&\qquad obj_2[move_2\langle l\rangle \triangleright \mathtt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : state_2\langle s_0\rangle : \{\}]\ |\!\} \vdash^{\sigma_1}_{jvm_{12}} \{\!|\ \ |\!\}
\end{aligned}
$$

$$
\begin{array}{cl}
\xrightarrow{5} & \{\!|\ library_2\ [\ c_{12}[D_1:\mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{32}[D_3:\mathsf{0}]:\mathsf{0}:\{\}\ ],\\
DefSol & \quad jvm_{22}\ [\ c_{14}[D_1:\mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{34}[D_3:\mathsf{0}]:\mathsf{0}:\sigma_2\ ]\\
 & \quad |\!\}\vdash^{\{\}} \{\!|\ obj_2.move_2\langle jvm_{22}\rangle\ |\!\}\\
 & \|\ \{\!|\ \ c_{13}[D_1:\mathsf{0}:\{\}],\ c_{23}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}:\{\}],\ c_{33}[D_3:\mathsf{0}:\{\}]\ |\!\}\vdash^{\sigma_1}_{jvm_{12}} \{\!|\ \ |\!\}\\
 & \|\ \{\!|\ \ move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle,\ f_2\langle\rangle | state_2\langle s\rangle \rhd P\ |\!\}\vdash^{\sigma_1}_{jvm_{12}obj_2} \{\!|\ state_2\langle s_0\rangle\ |\!\}
\end{array}
$$

$$
\begin{array}{cl}
\xrightarrow{6} & \{\!|\ library_2\ [\ c_{12}[D_1:\mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{32}[D_3:\mathsf{0}]:\mathsf{0}:\{\}\ ],\\
Comm & \quad jvm_{22}\ [\ c_{14}[D_1:\mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{34}[D_3:\mathsf{0}]:\mathsf{0}:\sigma_2\ ]\ |\!\}\vdash^{\{\}} \{\!|\ \ |\!\}\\
 & \|\ \{\!|\ \ c_{13}[D_1:\mathsf{0}:\{\}],\ c_{23}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}:\{\}],\ c_{33}[D_3:\mathsf{0}:\{\}]\ |\!\}\vdash^{\sigma_1}_{jvm_{12}} \{\!|\ \ |\!\}\\
 & \|\ \{\!|\ \ move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle,\ f_2\langle\rangle | state_2\langle s\rangle \rhd P\ |\!\}\vdash^{\sigma_1}_{jvm_{12}obj_2} \{\!|\ state_2\langle s_0\rangle,\ move_2\langle jvm_{22}\rangle\ |\!\}
\end{array}
$$

$$
\begin{array}{cl}
\xrightarrow{7} & \{\!|\ library_2\ [\ c_{12}[D_1:\mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{32}[D_3:\mathsf{0}]:\mathsf{0}:\{\}\ ],\\
Red & \quad jvm_{22}\ [\ c_{14}[D_1:\mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{34}[D_3:\mathsf{0}]:\mathsf{0}:\sigma_2\ ]\ |\!\}\vdash^{\{\}} \{\!|\ \ |\!\}\\
 & \|\ \{\!|\ \ c_{13}[D_1:\mathsf{0}:\{\}],\ c_{23}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}:\{\}],\ c_{33}[D_3:\mathsf{0}:\{\}]\ |\!\}\vdash^{\sigma_1}_{jvm_{12}} \{\!|\ \ |\!\}\\
 & \|\ \{\!|\ \ move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle,\ f_2\langle\rangle | state_2\langle s\rangle \rhd P\ |\!\}\vdash^{\sigma_1}_{jvm_{12}obj_2} \{\!|\ state_2\langle s_0\rangle,\ \mathtt{gostate}\langle jvm_{22},f_2\rangle\ |\!\}
\end{array}
$$

$$
\begin{array}{cl}
\xrightarrow{8} & \{\!|\ library_2\ [\ c_{12}[D_1:\mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{32}[D_3:\mathsf{0}]:\mathsf{0}:\{\}\ ],\\
SolDef & \quad jvm_{22}\ [\ c_{14}[D_1:\mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{34}[D_3:\mathsf{0}]:\mathsf{0}:\sigma_2\ ]\ |\!\}\vdash^{\{\}} \{\!|\ \ |\!\}\\
 & \|\ \{\!|\ \ c_{13}[D_1:\mathsf{0}:\{\}],\ c_{23}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}:\{\}],\ c_{33}[D_3:\mathsf{0}:\{\}],\\
 & \quad obj_2[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P : state_2\langle s_0\rangle\,|\,\mathtt{gostate}\langle jvm_{22},f_2\rangle:\{\}]\ |\!\}\vdash^{\sigma_1}_{jvm_{12}} \{\!|\ \ |\!\}
\end{array}
$$

$$
\begin{array}{cl}
\xrightarrow{9} & \{\!|\ library_2\ [\ c_{12}[D_1:\mathsf{0}]\ \wedge\ c_{22}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{32}[D_3:\mathsf{0}]:\mathsf{0}:\{\}\ ],\\
Contract & \quad jvm_{22}\ [\ c_{14}[D_1:\mathsf{0}]\ \wedge\ c_{24}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}]\ \wedge\ c_{34}[D_3:\mathsf{0}]:\mathsf{0}:\sigma_2\ ]\ |\!\}\vdash^{\{\}} \{\!|\ \ |\!\}\\
 & \|\ \{\!|\ \ c_{13}[D_1:\mathsf{0}:\{\}],\ c_{23}[move_2\langle l\rangle \rhd \mathtt{gostate}\langle l,f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \rhd P:\mathsf{0}:\{\}],\ c_{33}[D_3:\mathsf{0}:\{\}],\\
 & \quad obj_2[\,\grave{}\,c_{22} : state_2\langle s_0\rangle\,|\,\mathtt{gostate}\langle jvm_{22},f_2\rangle:\{\}]\ |\!\}\vdash^{\sigma_1}_{jvm_{12}} \{\!|\ \ |\!\}
\end{array}
$$

$$\begin{aligned}
\xrightarrow[SolDef]{10}\quad & \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ],\\
& \quad jvm_{12}\ [\ c_{13}[D_1 : \mathsf{0} : \{\}] \wedge c_{23}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}] \wedge c_{33}[D_3 : \mathsf{0} : \{\}] \wedge\\
& \qquad\qquad obj_2[\lq c_{22} : state_2\langle s_0\rangle \,|\, \texttt{gostate}\langle jvm_{22}, f_2\rangle : \{\}] : \mathsf{0} : \sigma_1\ ],\\
& \quad jvm_{22}\ [\ c_{14}[D_1 : \mathsf{0}] \wedge c_{24}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}] \wedge c_{34}[D_3 : \mathsf{0}] : \mathsf{0} : \sigma_2\ ]\ |\!\}\vdash^{\{\}} \{\!|\quad|\!\}
\end{aligned}$$

$$\begin{aligned}
\xrightarrow[MoveS]{11}\quad & \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ],\\
& \quad jvm_{12}\ [\ c_{13}[D_1 : \mathsf{0} : \{\}] \wedge c_{23}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}] \wedge c_{33}[D_3 : \mathsf{0} : \{\}] : \mathsf{0} : \sigma_1\ ],\\
& \quad jvm_{22}\ [\ c_{14}[D_1 : \mathsf{0}] \wedge c_{24}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}] \wedge c_{34}[D_3 : \mathsf{0}] \wedge\\
& \qquad\qquad obj_2[\lq c_{22} : state_2\langle s_0\rangle \,|\, f_2\langle\rangle : \{\}] : \mathsf{0} : \sigma_2\ ]\ |\!\}\vdash^{\{\}} \{\!|\quad|\!\}
\end{aligned}$$

$$\begin{aligned}
\xrightarrow[DefSol]{12}\quad & \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ],\\
& \quad jvm_{12}\ [\ c_{13}[D_1 : \mathsf{0} : \{\}] \wedge c_{23}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}] \wedge c_{33}[D_3 : \mathsf{0} : \{\}] : \mathsf{0} : \sigma_1\ ]\ |\!\}\vdash^{\{\}} \{\!|\quad|\!\}\\
& \|\ \{\!|\ \ c_{14}[D_1 : \mathsf{0} : \{\}]\,,\ c_{24}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}]\,,\ c_{34}[D_3 : \mathsf{0} : \{\}]\,,\\
& \qquad obj_2[\lq c_{22} : state_2\langle s_0\rangle \,|\, f_2\langle\rangle : \{\}]\ |\!\}\vdash^{\sigma_2}_{jvm_{22}} \{\!|\quad|\!\}
\end{aligned}$$

$$\begin{aligned}
\xrightarrow[Copy]{13}\quad & \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ],\\
& \quad jvm_{12}\ [\ c_{13}[D_1 : \mathsf{0} : \{\}] \wedge c_{23}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}] \wedge c_{33}[D_3 : \mathsf{0} : \{\}] : \mathsf{0} : \sigma_1\ ]\ |\!\}\vdash^{\{\}} \{\!|\quad|\!\}\\
& \|\ \{\!|\ \ c_{14}[D_1 : \mathsf{0} : \{\}]\,,\ c_{24}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}]\,,\ c_{34}[D_3 : \mathsf{0} : \{\}]\,,\\
& \qquad obj_2[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : state_2\langle s_0\rangle \,|\, f_2\langle\rangle : \{\}]\ |\!\}\vdash^{\sigma_2}_{jvm_{22}} \{\!|\quad|\!\}
\end{aligned}$$

$$\begin{aligned}
\xrightarrow[DefSol]{14}\quad & \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ],\\
& \quad jvm_{12}\ [\ c_{13}[D_1 : \mathsf{0} : \{\}] \wedge c_{23}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}] \wedge c_{33}[D_3 : \mathsf{0} : \{\}] : \mathsf{0} : \sigma_1\ ]\ |\!\}\vdash^{\{\}} \{\!|\quad|\!\}\\
& \|\ \{\!|\ \ c_{14}[D_1 : \mathsf{0} : \{\}]\,,\ c_{24}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}]\,,\ c_{34}[D_3 : \mathsf{0} : \{\}]\ |\!\}\vdash^{\sigma_2}_{jvm_{22}} \{\!|\quad|\!\}\\
& \|\ \{\!|\ \ move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle\ ,\ f_2\langle\rangle | state_2\langle s\rangle \triangleright P\ |\!\}\vdash^{\sigma_2}_{jvm_{22}obj_2} \{\!|\ state_2\langle s_0\rangle \,|\, f_2\langle\rangle\ |\!\}
\end{aligned}$$

$$\begin{aligned}
\xrightarrow[Red]{15}\quad & \{\!|\ library_2\ [\ c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] : \mathsf{0} : \{\}\ ],\\
& \quad jvm_{12}\ [\ c_{13}[D_1 : \mathsf{0} : \{\}] \wedge c_{23}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}] \wedge c_{33}[D_3 : \mathsf{0} : \{\}] : \mathsf{0} : \sigma_1\ ]\ |\!\}\vdash^{\{\}} \{\!|\quad|\!\}\\
& \|\ \{\!|\ \ c_{14}[D_1 : \mathsf{0} : \{\}]\,,\ c_{24}[move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle \wedge f_2\langle\rangle | state_2\langle s\rangle \triangleright P : \mathsf{0} : \{\}]\,,\ c_{34}[D_3 : \mathsf{0} : \{\}]\ |\!\}\vdash^{\sigma_2}_{jvm_{22}} \{\!|\quad|\!\}\\
& \|\ \{\!|\ \ move_2\langle l\rangle \triangleright \texttt{gostate}\langle l, f_2\rangle\ ,\ f_2\langle\rangle | state_2\langle s\rangle \triangleright P\ |\!\}\vdash^{\sigma_2}_{jvm_{22}obj_2} \{\!|\ P\{s_0/s\}\ |\!\}
\end{aligned}$$

D.2 Ex.2 Dynamic Class Downloading

The following shows the necessary steps when performing the computations for the example in section 8.2. The starting point is the chemical solution $\{\!|\ \ |\!\} \vdash^{\{\}} \{\!|P|\!\}$ where P is the process from the example. The solution will be heated and cooled as necessary in order to make the molecules react. The listing below describes each step in words; the development of the chemical solution can be seen on the following pages. The heating/cooling rules $Join$, $Null$, $DefAnd$, $DComm$, $DAssoc$, $DefNull$, and $DefNull2$ will be applied to the solution whenever needed; all other rules will be mentioned explicitly when they are applied.

1. The solution is heated according to the rules Def and $DefLoc$. This introduces empty substitutions into each location definition and introduces new names for all the defined variables. The names defined inside the local definition construct in location $http$ do not belong to the defined variables and will not become substituted. The applied substitution is:
$$\begin{aligned}\sigma_{dv} = \{&c_{12}/c_1,\ c_{22}/c_2,\ c_{32}/c_3,\ ext_2/ext,\ http_2/http,\ jvm_2/jvm,\ obj_2/obj,\\ &classloader_2/classloader,\ fetch_2/fetch,\ get_2/get,\ state_2/state\}\end{aligned}$$
2. The solution is heated according to the $Copy$ rule. The definitions from ext_2 are copied to $http_2$ and new names are introduced for the copied locations. The connection between original location names and the names of the copies are stored in the substitution:
$$\sigma_1 = \{c_{13}/c_{12},\ c_{23}/c_{22},\ c_{33}/c_{32}\}$$
3. The location definition of $http_2$ is heated to solution form using the rule $DefSol$. This will make $http_2$ ready to accept requests for class definitions. The $DefSol$ rule and the $DefLoc$ rule will also be used on jvm_2 and next on $classloader_2$ to make these locations ready for computations. The location obj_2 cannot be heated to solution form because it isn't fully expanded. The expansion of the location is not possible yet because there is no local instance of the class c_{22} ($g_2(\grave{}c_{22}, \{\}) = c_{22}$ so the $Copy$ rule cannot be applied).
4. The message from jvm_2 to $classloader_2$ is sent using the $Comm$ rule.
5. The Red rule is applied on the solution $\vdash_{jvm_2 classloader_2}$. The reduction builds on the substitution $\sigma_{rv} = \{c_{22}/class,\ http_2/cb\}$
6. The message from $classloader_2$ to $http_2$ is sent using the $Comm$ rule.
7. The Red rule is applied on the solution $\vdash_{http_2}$. The reduction builds on the substitution $\sigma_{rv} = \{c_{22}/class,\ jvm_2/l\}$
8. The solution $\vdash_{http_2}$ is heated according to the Def rule. New names are introduced for the defined variables using the substitution $\sigma_{dv} = \{newc_2/newc\}$. An empty substitution is introduced into the location definition for $newc_2$ using the $DefLoc$ rule.
9. The location definition $newc_2$ is heated according to the $Copy$ rule (neither $Subst$ nor $Move$ can be applied until this has happened). It is the definitions from c_{23} that will be copied since $g_1(\grave{}c_{22}, \sigma_1) = c_{23}$.

10. The solutions $\vdash_{jvm_2classloader_2}$, $\vdash_{jvm_2}$ and $\vdash_{http_2}$ are cooled according to the $SolDef$ rule. This will allow the $Move$ or the $Subst$ rule to be applied.
11. The location $newc_2$ is moved from $http_2$ to jvm_2 using the $Move$ rule (if the $Subst$ rule were applied first, then it wouldn't be possible to move the $newc_2$ location, and the solution would have to be cooled again with the $Subst$ rule).
12. The $Subst$ rule is applied to jvm_2. This results in $newc_2$ being seen as a local instance of c_{22} inside jvm_2.
13. The location obj_2 can now be expanded using the fetched class.

$$
\begin{array}{l}
\{\!|\ \ |\!\} \vdash^{\{\}} \{\!|\ \mathtt{def}\quad ext[\ c_1[D_1 : 0] \wedge c_2[D_2 : 0] \wedge c_3[D_3 : 0] \wedge \ldots : 0\ ] \\
\qquad\qquad \wedge\ \ http[\ ext\ \wedge\ fetch\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ `class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); 0]\ \mathtt{in}\ 0 : 0\ ] \\
\qquad\qquad \wedge\ \ jvm[\ classloader[\ get\langle class, cb\rangle \rhd cb.fetch\langle class, jvm\rangle\ ] \wedge obj[\ `c_2 : state\langle s\rangle\ ] \\
\qquad\qquad\qquad\quad :\ classloader.get\langle c_2, http\rangle\ ] \\
\qquad\quad\ \mathtt{in}\quad 0 \hfill |\!\}
\end{array}
$$

$$
\begin{array}{ll}
\underset{Def}{\overset{1}{\rightharpoonup}}{}^{*} &
\begin{array}{l}
\{\!|\ \ ext_2[\ c_{12}[D_1 : 0] \wedge c_{22}[D_2 : 0] \wedge c_{32}[D_3 : 0] \wedge \ldots : 0 : \{\}\ ], \\
\quad http_2[\ ext_2\ \wedge\ fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ `class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); 0]\ \mathtt{in}\ 0 : 0 : \{\}\ ], \\
\quad jvm_2[\ classloader_2[\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ ] \wedge obj_2[\ `c_{22} : state_2\langle s\rangle\ ] \\
\qquad\quad :\ classloader_2.get_2\langle c_{22}, http_2\rangle : \{\}\ ] \hfill |\!\} \vdash^{\{\}} \{\!|\ \ |\!\}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\underset{Copy}{\overset{2}{\rightharpoonup}} &
\begin{array}{l}
\{\!|\ \ ext_2[\ c_{12}[D_1 : 0] \wedge c_{22}[D_2 : 0] \wedge c_{32}[D_3 : 0] \wedge \ldots : 0 : \{\}\ ], \\
\quad http_2[\ c_{13}[D_1 : 0] \wedge c_{23}[D_2 : 0] \wedge c_{33}[D_3 : 0] \wedge \ldots \wedge \\
\qquad\quad fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ `class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); 0]\ \mathtt{in}\ 0 : 0 : \sigma_1\ ], \\
\quad jvm_2[\ classloader_2[\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ ] \wedge obj_2[\ `c_{22} : state_2\langle s\rangle\ ] \\
\qquad\quad :\ classloader_2.get_2\langle c_{22}, http_2\rangle : \{\}\ ] \hfill |\!\} \vdash^{\{\}} \{\!|\ \ |\!\}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\underset{DefSol}{\overset{3}{\rightharpoonup}}{}^{*} &
\begin{array}{l}
\quad \{\!|\ ext_2[\ c_{12}[D_1 : 0] \wedge c_{22}[D_2 : 0] \wedge c_{32}[D_3 : 0] \wedge \ldots : 0 : \{\}\ ]\ |\!\} \vdash^{\{\}} \{\!|\ \ |\!\} \\
\parallel \{\!|\ c_{13}[D_1 : 0],\ c_{23}[D_2 : 0],\ c_{33}[D_3 : 0],\ \ldots\ , \\
\qquad fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ `class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); 0]\ \mathtt{in}\ 0\ |\!\}\ \vdash^{\sigma_1}_{http_2}\ \{\!|\ \ |\!\} \\
\parallel \{\!|\ obj_2[\ `c_{22} : state_2\langle s\rangle : \{\}\ ]\ |\!\}\ \vdash^{\{\}}_{jvm_2}\ \{\!|\ classloader_2.get_2\langle c_{22}, http_2\rangle\ |\!\} \\
\parallel \{\!|\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ |\!\}\ \vdash^{\{\}}_{jvm_2 classloader_2}\ \{\!|\ \ |\!\}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\underset{Comm}{\overset{4}{\rightarrow}} &
\begin{array}{l}
\quad \{\!|\ ext_2[\ c_{12}[D_1 : 0] \wedge c_{22}[D_2 : 0] \wedge c_{32}[D_3 : 0] \wedge \ldots : 0 : \{\}\ ]\ |\!\} \vdash^{\{\}} \{\!|\ \ |\!\} \\
\parallel \{\!|\ c_{13}[D_1 : 0],\ c_{23}[D_2 : 0],\ c_{33}[D_3 : 0],\ \ldots\ , \\
\qquad fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ `class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); 0]\ \mathtt{in}\ 0\ |\!\}\ \vdash^{\sigma_1}_{http_2}\ \{\!|\ \ |\!\} \\
\parallel \{\!|\ obj_2[\ `c_{22} : state_2\langle s\rangle : \{\}\ ]\ |\!\}\ \vdash^{\{\}}_{jvm_2}\ \{\!|\ \ |\!\} \\
\parallel \{\!|\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ |\!\}\ \vdash^{\{\}}_{jvm_2 classloader_2}\ \{\!|\ get_2\langle c_{22}, http_2\rangle\ |\!\}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\xrightarrow[\textit{Red}]{5} & \{\!|\; ext_2[\; c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[D_2 : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] \wedge \ldots : \mathsf{0} : \{\}\;] \;|\!\} \vdash^{\{\}} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; c_{13}[D_1 : \mathsf{0}]\,,\, c_{23}[D_2 : \mathsf{0}]\,,\, c_{33}[D_3 : \mathsf{0}],\; \ldots\;, \\
& \quad fetch_2\langle class, l\rangle \rhd \;\texttt{def}\; newc[\;\grave{}class : \texttt{subst}\langle class\rangle \,|\, \texttt{go}(l); \mathsf{0}]\; \texttt{in}\; \mathsf{0} \;|\!\} \vdash^{\sigma_1}_{http_2} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; obj_2[\;\grave{}c_{22} : state_2\langle s\rangle : \{\}\;] \;|\!\} \vdash^{\{\}}_{jvm_2} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle \;|\!\} \vdash^{\{\}}_{jvm_2 classloader_2} \{\!|\; http_2.fetch_2\langle c_{22}, jvm_2\rangle \;|\!\}
\end{array}
$$

$$
\begin{array}{ll}
\xrightarrow[\textit{Comm}]{6} & \{\!|\; ext_2[\; c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[D_2 : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] \wedge \ldots : \mathsf{0} : \{\}\;] \;|\!\} \vdash^{\{\}} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; c_{13}[D_1 : \mathsf{0}]\,,\, c_{23}[D_2 : \mathsf{0}]\,,\, c_{33}[D_3 : \mathsf{0}],\; \ldots\;, \\
& \quad fetch_2\langle class, l\rangle \rhd \;\texttt{def}\; newc[\;\grave{}class : \texttt{subst}\langle class\rangle \,|\, \texttt{go}(l); \mathsf{0}]\; \texttt{in}\; \mathsf{0} \;|\!\} \vdash^{\sigma_1}_{http_2} \{\!|\; fetch_2\langle c_{22}, jvm_2\rangle \;|\!\} \\
& \parallel \{\!|\; obj_2[\;\grave{}c_{22} : state_2\langle s\rangle : \{\}\;] \;|\!\} \vdash^{\{\}}_{jvm_2} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle \;|\!\} \vdash^{\{\}}_{jvm_2 classloader_2} \{\!|\;\; |\!\}
\end{array}
$$

$$
\begin{array}{ll}
\xrightarrow[\textit{Red}]{7} & \{\!|\; ext_2[\; c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[D_2 : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] \wedge \ldots : \mathsf{0} : \{\}\;] \;|\!\} \vdash^{\{\}} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; c_{13}[D_1 : \mathsf{0}]\,,\, c_{23}[D_2 : \mathsf{0}]\,,\, c_{33}[D_3 : \mathsf{0}],\; \ldots\;, \\
& \quad fetch_2\langle class, l\rangle \rhd \;\texttt{def}\; newc[\;\grave{}class : \texttt{subst}\langle class\rangle \,|\, \texttt{go}(l); \mathsf{0}]\; \texttt{in}\; \mathsf{0} \\
& \quad |\!\} \vdash^{\sigma_1}_{http_2} \{\!|\; \texttt{def}\; newc[\;\grave{}c_{22} : \texttt{subst}\langle c_{22}\rangle \,|\, \texttt{go}(jvm_2); \mathsf{0}]\; \texttt{in}\; \mathsf{0} \;|\!\} \\
& \parallel \{\!|\; obj_2[\;\grave{}c_{22} : state_2\langle s\rangle : \{\}\;] \;|\!\} \vdash^{\{\}}_{jvm_2} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle \;|\!\} \vdash^{\{\}}_{jvm_2 classloader_2} \{\!|\;\; |\!\}
\end{array}
$$

$$
\begin{array}{ll}
\underset{\textit{Def}^*}{\overset{8}{\searrow}} & \{\!|\; ext_2[\; c_{12}[D_1 : \mathsf{0}] \wedge c_{22}[D_2 : \mathsf{0}] \wedge c_{32}[D_3 : \mathsf{0}] \wedge \ldots : \mathsf{0} : \{\}\;] \;|\!\} \vdash^{\{\}} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; c_{13}[D_1 : \mathsf{0}]\,,\, c_{23}[D_2 : \mathsf{0}]\,,\, c_{33}[D_3 : \mathsf{0}],\; \ldots\;, \\
& \quad fetch_2\langle class, l\rangle \rhd \;\texttt{def}\; newc[\;\grave{}class : \texttt{subst}\langle class\rangle \,|\, \texttt{go}(l); \mathsf{0}]\; \texttt{in}\; \mathsf{0}, \\
& \quad newc_2[\;\grave{}c_{22} : \texttt{subst}\langle c_{22}\rangle \,|\, \texttt{go}(jvm_2); \mathsf{0} : \{\}\;] \;|\!\} \vdash^{\sigma_1}_{http_2} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; obj_2[\;\grave{}c_{22} : state_2\langle s\rangle : \{\}\;] \;|\!\} \vdash^{\{\}}_{jvm_2} \{\!|\;\; |\!\} \\
& \parallel \{\!|\; get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle \;|\!\} \vdash^{\{\}}_{jvm_2 classloader_2} \{\!|\;\; |\!\}
\end{array}
$$

$$\xrightarrow[Copy]{9} \quad \{\!|\ ext_2[\ c_{12}[D_1 : \mathtt{0}] \wedge c_{22}[D_2 : \mathtt{0}] \wedge c_{32}[D_3 : \mathtt{0}] \wedge \ldots : \mathtt{0} : \{\}\ ]\ |\!\} \vdash^{\{\}} \{\!|\ \ |\!\}$$
$$\parallel \{\!|\ c_{13}[D_1 : \mathtt{0}]\ ,\ c_{23}[D_2 : \mathtt{0}]\ ,\ c_{33}[D_3 : \mathtt{0}],\ \ldots\ ,$$
$$fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ \grave{}class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); \mathtt{0}]\ \mathtt{in}\ \mathtt{0},$$
$$newc_2[\ D_2 : \mathtt{subst}\langle c_{22}\rangle \,|\, \mathtt{go}(jvm_2); \mathtt{0} : \{\}\ ]\ |\!\}\ \vdash^{\sigma_1}_{http_2}\ \{\!|\ \ |\!\}$$
$$\parallel \{\!|\ obj_2[\ \grave{}c_{22} : state_2\langle s\rangle : \{\}\ ]\ |\!\}\ \vdash^{\{\}}_{jvm_2}\ \{\!|\ \ |\!\}$$
$$\parallel \{\!|\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ |\!\}\ \vdash^{\{\}}_{jvm_2 classloader_2}\ \{\!|\ \ |\!\}$$

$$\xrightarrow[SolDef]{10^*} \quad \{\!|\ \ ext_2[\ c_{12}[D_1 : \mathtt{0}] \wedge c_{22}[D_2 : \mathtt{0}] \wedge c_{32}[D_3 : \mathtt{0}] \wedge \ldots : \mathtt{0} : \{\}\ ],$$
$$http_2[\ c_{13}[D_1 : \mathtt{0}] \wedge c_{23}[D_2 : \mathtt{0}] \wedge c_{33}[D_3 : \mathtt{0}] \wedge \ \ldots \ \wedge$$
$$fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ \grave{}class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); \mathtt{0}]\ \mathtt{in}\ \mathtt{0} \wedge$$
$$newc_2[\ D_2 : \mathtt{subst}\langle c_{22}\rangle \,|\, \mathtt{go}(jvm_2); \mathtt{0} : \{\}\ ] : \mathtt{0} : \sigma_1\ ],$$
$$jvm_2[\ classloader_2[\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ ] : \mathtt{0} : \{\}\ ] \wedge obj_2[\ \grave{}c_{22} : state_2\langle s\rangle : \{\}\ ]\ |\!\} \vdash^{\{\}} \{\!|\ \ |\!\}$$

$$\xrightarrow[Move]{11} \quad \{\!|\ \ ext_2[\ c_{12}[D_1 : \mathtt{0}] \wedge c_{22}[D_2 : \mathtt{0}] \wedge c_{32}[D_3 : \mathtt{0}] \wedge \ldots : \mathtt{0} : \{\}\ ],$$
$$http_2[\ c_{13}[D_1 : \mathtt{0}] \wedge c_{23}[D_2 : \mathtt{0}] \wedge c_{33}[D_3 : \mathtt{0}] \wedge \ \ldots \ \wedge$$
$$fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ \grave{}class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); \mathtt{0}]\ \mathtt{in}\ \mathtt{0} : \mathtt{0} : \sigma_1\ ],$$
$$jvm_2[\ classloader_2[\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ ] \wedge$$
$$obj_2[\ \grave{}c_{22} : state_2\langle s\rangle : \{\}\ ]\ \wedge\ newc_2[\ D_2 : \mathtt{subst}\langle c_{22}\rangle : \{\}\ ] : \mathtt{0} : \{\}\ ] \qquad |\!\} \vdash^{\{\}} \{\!|\ \ |\!\}$$

$$\xrightarrow[Subst]{12} \quad \{\!|\ \ ext_2[\ c_{12}[D_1 : \mathtt{0}] \wedge c_{22}[D_2 : \mathtt{0}] \wedge c_{32}[D_3 : \mathtt{0}] \wedge \ldots : \mathtt{0} : \{\}\ ],$$
$$http_2[\ c_{13}[D_1 : \mathtt{0}] \wedge c_{23}[D_2 : \mathtt{0}] \wedge c_{33}[D_3 : \mathtt{0}] \wedge \ \ldots \ \wedge$$
$$fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ \grave{}class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); \mathtt{0}]\ \mathtt{in}\ \mathtt{0} : \mathtt{0} : \sigma_1\ ],$$
$$jvm_2[\ classloader_2[\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ ] \wedge$$
$$obj_2[\ \grave{}c_{22} : state_2\langle s\rangle : \{\}\ ]\ \wedge\ newc_2[\ D_2 : \mathtt{0} : \{\}\ ] : \mathtt{0} : \{newc_2/c_{22}\}\ ] \qquad |\!\} \vdash^{\{\}} \{\!|\ \ |\!\}$$

$$\xrightarrow[Copy3]{13} \quad \{\!|\ \ ext_2[\ c_{12}[D_1 : \mathtt{0}] \wedge c_{22}[D_2 : \mathtt{0}] \wedge c_{32}[D_3 : \mathtt{0}] \wedge \ldots : \mathtt{0} : \{\}\ ],$$
$$http_2[\ c_{13}[D_1 : \mathtt{0}] \wedge c_{23}[D_2 : \mathtt{0}] \wedge c_{33}[D_3 : \mathtt{0}] \wedge \ \ldots \ \wedge$$
$$fetch_2\langle class, l\rangle \rhd\ \mathtt{def}\ newc[\ \grave{}class : \mathtt{subst}\langle class\rangle \,|\, \mathtt{go}(l); \mathtt{0}]\ \mathtt{in}\ \mathtt{0} : \mathtt{0} : \sigma_1\ ],$$
$$jvm_2[\ classloader_2[\ get_2\langle class, cb\rangle \rhd cb.fetch_2\langle class, jvm_2\rangle\ ] \wedge$$
$$obj_2[\ D_2 : state_2\langle s\rangle : \{\}\ ]\ \wedge\ newc_2[\ D_2 : \mathtt{0} : \{\}\ ] : \mathtt{0} : \{newc_2/c_{22}\}\ ] \qquad |\!\} \vdash^{\{\}} \{\!|\ \ |\!\}$$