VRIJE UNIVERSITEIT BRUSSEL
FACULTY OF SCIENCES ~ COMPUTER SCIENCE DEPARTMENT

Creation of an Intelligent Concurrency Adaptor in order to mediate the Differences between Conflicting Concurrency Interfaces

Werner Van Belle

August 2001 – May 2003

Adviser: Prof. Dr. Theo D'Hondt
co-Adviser: Dr. Tom Mens

Werner Van Belle
Programming Technology Lab (PROG)
Department Computer Science (DINF)
Vrije Universiteit Brussel (VUB)
Pleinlaan 2
1050 Brussels
Belgium

e-mail: Werner.van.belle@vub.ac.be
phone: +32 486 68 84 48
fax: +32 2 629 35 25

The author of this dissertation was funded by the FKFO during 1 year, during which he developed the Borg virtual machine. After that he worked for 4 years on the SEESCOA Project (funded by the IWT) where he helped developing a distributed real-time system.

Abstract

THE DISSERTATION YOU ARE ABOUT TO READ, tries to solve one of the more prominent problems within open distributed systems namely: concurrency management between components written by different manufacturers. All too often, the concurrency strategy provided or required by a component is badly documented and rarely matches the concurrency strategy provided or required by another component. Whenever this happens there is a concurrency interface conflict. Solving these conflicts requires a substantial amount of resources with respect to software engineering: the time necessary to understand the problem, the time necessary to solve the problem, and above all the resources towards maintaining a working concurrency strategy that mediates between the different components. Indeed, in open distributed systems, components can be updated without prior notification and without guarantees that the new interface is backward compatible. Such updates can range from syntactic modifications over slight semantic differences to completely new concurrency strategies. For example, changing a nested locking strategy to a non-nested locking strategy or changing a non-blocking server to work synchronously.

In order to solve the problem of conflicting concurrency interfaces we will create a concurrency adaptor that resolves incompatibilities between incompatible concurrency strategies. We do this in two steps: first we require a certain amount of extra information to be present: every provided and required interface should be documented by means of colored Petri-nets and certain checkpoints are to be placed in the code to check the liveness.

Second, we construct a concurrency adaptor that can be placed between the different communicating components. This is done by means of a hybrid approach: first the adaptor will try to gain freedom by bypassing all the existing concurrency strategies. For a client a stub concurrency interface is generated that will keep the client alive. For a server a stub concurrency interface is generated that will allow anything to happen; in essence bypassing the concurrency strategy entirely. The concurrency adaptor is finished by plugging in an existing, formally guaranteed to work concurrency strategy between the two stub concurrency interfaces.

Bypassing a server's behavior is achieved by means of a runtime formal deduction. Given the current state of the Petri-net and the required state a prolog program deduces what should happen. Bypassing a clients behavior is achieved with a reinforcement learning algorithm that maximizes the reward it receives from the component itself. The rewards are based on checkpoints as specified by the component itself.

When placing a guaranteed to work concurrency strategy between the different stub concurrency-interfaces, we need a meta-protocol that is understood by this central concurrency strategy. This meta-protocol specifies which resources are present and which locking/unlocking operations can work upon them. The meta-protocol is deduced entirely from the Petri-nets involved.

The approach presented in this dissertation provides a substantial added value to the programmer of components in open distributed systems. He now only needs to specify what he requires or provides as a concurrency strategy within his component. He no longer needs to take into account the concurrency strategy offered by other components. This might reduce development and maintenance time drastically. A second advantage of using Petri-nets is that interfaces are not only documented, but that this information can be verified automatically: whenever necessary the formal specification can be tested against the actual working of a component.

keywords: *distributed systems, mobile multi-agent systems, Petri-nets, concurrency strategies, reinforcement learning, component based development.*

Contents

List of Algorithms

List of Figures

List of Tables

Acknowledgments

I would like to thank my closest supervisors TOM TOURWÉ, TOM MENS and THEO D'HONDT for guiding me in my work. Their to-the-point remarks have substantially helped clearing up the text.

Aside from my three immediate supervisors, I thank WOLFGANG DEMEUTER, JOHAN FABRY and DIRK VAN DEUN, who spent many hours together with me behind a text editor to improve the readability of this text. I enjoyed being in the same group with them and I hope that in the near future they will find the time and resources to finish their own work. If the time comes, I will be glad to be of assistance.

Furthermore, I thank JESSIE DEDECKER and JOHAN BRICHAU for proofreading this dissertation.

I thank the members of the jury, KOEN DE BOSSCHERE (University Ghent, Group PARIS), ANN NOWÉ (University Brussels, Group CoMo), DIRK VERMEIR (University Brussels, TINF), MADS NYGÅRD (IDI/NTNU), YOLANDE BERBERS (KUL/DISTRINET) for their feedback.

I thank MICHIEL RONSSE, DAVID URTING, TOM TOUTENEL, PETER RIGOLE, MARK CHRISTIAENS, KRIS LUYTEN, JAN VAN DEN BERGH, DENNIS WAGELAER and CHRIS VANDERVELPEN. I very much enjoyed working together with them at the SEESCOA research project the past 4 years.

I would also like to thank the people that have been helping on the Borg Project: JOHAN FABRY, DIRK VAN DEUN and KARSTEN VERELST.

And last, I thank my adviser THEO D'HONDT for giving me the freedom and possibilities to finish this work.

General Glossary

API	Application Program Interface: the syntactical one way description of an interface. An API typically says which functions can be called with which arguments and what will be returned. An API describes in a formal way how functions should be called. Together with this it can also describe what a function does in an informal way. We do not consider informal documentation to be part of the API.
asynchronous	In this dissertation we will use the term asynchronous to specify the *behavior* of a component that sends messages, without waiting for the return value from another component. This should not to be confused with non-blocking. It is clear that an asynchronous working client uses non-blocking primitives, however, not all non-blocking messages will result in an asynchronous working component.
behavior	The behavior of a component is the result of its internal programming. Typically the behavior of a component can be seen by looking at the message flow over its interface, however the exact program describing the behavior exactly is unknown.
blocking	Call and wait for return value.
call	a call is sending a request to another object or component and *waiting* for the answer. Calls always return a value (which possibly can be void)
component	A *component* is a process that can *only* communicate with other components by means of messages. Components do not share code and or data.
client	in this dissertation, a client is a component that requires a certain concurrency strategy from another component to be able to do its work.
component system	To be able to run components one needs the infrastructure that allows communication, naming and multitasking. This infrastructure is called the *component system*.
conflicting interface	Given a) two interfaces and b) an external (not specified in the interface) agreement on the expected behavior between the involved components, then these two interfaces are in conflict if this external behavior cannot be executed on the link between the two interfaces. For instance, a conflict might arise when a message cannot be understood by one of the involved components, or if a message will exhibit other behavior than the expected behavior, or simply if a sequence of actions leads to corruption of the internal state of the involved components.
co-NP	A problem p is co-NP if $\neg p$ is NP. co-NP problems often seem harder than NP problems, however it is still an open question whether co-NP = NP.

event	See *message.*
EXPTIME	A problem of size n is EXPTIME if there is a constant c and the time to solve the problem takes at most 2^{n^c}.
EXPTIME-complete	A problem is EXPTIME-complete if it is EXPTIME and other EXPTIME problems are reducible to it. Essentially this declares a class of problems which can be solved within exponential time. It is known that EXPTIME $\neq$ PTIME.
interface	An interface is a connection-point to a component. It is the only means to alter the internal state and/or the behavior of the component. In the component system interfaces are embodied under the form of ports which can be connected to other components.
machine	The term machine is used to refer to Turing machines.
message	A piece of data used to communicate between components. A message can be sent to another component or it can be received. Message never share data. Also called *event*.
multi-multi	A multi on multi interface conflict is a conflict between a number of interface providing components and a number of interface requiring components. This typically occurs in peer to peer systems.
non-blocking	Send a message and don't wait for an answer.
non-deterministic	A machine is non-deterministic if it may execute different branches parallel on the same input.
NP	A problem is NP if it can be solved in polynomial time by a nondeterministic machine.
NP-complete	A problem is NP-complete if other NP-complete problems can be reduced to it.
one-one	A one on one interface conflict is a conflict between two components. Typically one component is a server, the other component is a client.
one-multi	A one on multi interface conflict is a conflict between one server component and a number of client components.
process	A process is a program which is running in a separate code and data space. Processes can have multiple threads.
P	A problem of size n is P if there is a constant c and the time to solve the problem is at most n^c.
P-complete	A problem is P-complete if it is in P and other P-problems are reducible to it.
P-hard	A problem is P-hard if itself is not necessarily in P but other P-problems may reduce to it.
server	In this dissertation, a server is a component that provides a concurrency strategy towards a number of clients.
thread	A thread is an execution environment that runs in a process. A process can consist of multiple threads.

whiteboard	A typical application in distributed systems where many users join a place to discuss. Every user has a view on the shared data and can modify this as necessary. All other users will see those changes. A whiteboard is a tool in a group discussion system.

Chapter 1

Introduction

WITH THE ADVENT OF THE INTERNET, DISTRIBUTED SYSTEMS seem to have become an ubiquitous phenomenon. Ever more organizations that were are already present on the Internet with a website start to feel the need to extend the services they offer. These vary from simple scripting applications (making their website a little more intelligent and appealing to the end-user), to applications that are literally distributed and in which the cooperating nodes are technically equally powerful and thus abandon the traditional thin-client/fat-server point of view.

Unfortunately the construction of such distributed applications is far from trivial. Not only does one need to take into account the possibility of failures, all distributed systems are inherently concurrent systems. And the construction of concurrent systems is widely accepted to be an extremely delicate and complex problem. One of the complexities lies in the selection and correct realization of a suitable *concurrency strategy* in a certain context. Such a strategy is responsible for the correct cooperation of different concurrently operating components. The reason for the complexity stems from the fact that the actual concurrency strategy cannot be localized in one of the operating components but spans the entire application. Hence, the concurrency strategy is by definition an agreement between the concurrently operating components. Its implementation will thus leave traces in all those components. In other words, two strategies needs to be compatible or otherwise the software will not work. This problem is further aggravated in the context of open distributed systems, i.e. systems in which not all components are under control of the same entity, and for which the components' implementations have no control over each other.

The goal of this dissertation is to create *an adaptor* between the interfaces of components that are in essence compatible with each other, but fail to cooperate simply because of conflicts between the concurrency strategies they employ. Concurrency strategies are in conflict if one of the involved components fails to be alive and/or suffers from data races, i.e. concurrency that results in data corruption. The adaptor we will create is responsible for mediating the differences between the different concurrency interfaces in such a way that both components can communicate in a way that makes sense. Of course, the adaptor will not affect the core functionality of the components themselves.

In this dissertation, we will illustrate that creating such an intelligent concurrency adaptor is possible. Throughout the thesis it should be kept in mind that we will not be solving the inherent problems of distributed and concurrent systems. Instead they form the setting in which we conducted our research.

1.1 Motivation

THE PROBLEM OF CONCURRENCY CONFLICTS outlined above is an instance of what one might generally call interface conflicts in distributed systems. With the current state of the art in connection technology these interface conflicts emerge in apparently different situations which together form the broader context of our work. In the sections to come we will outline the following three different points:

1. A first context in which the problem emerges is the one of *standards in open distributed systems*. In general, these are defined by companies and are a means to stay in control of how their software is used. The interfaces used and provided by companies reflect their interest in data exchange and cooperation. Companies with the ability to generate good interfaces fast and with the ability to create interfaces that adapt to their environment have a strategic advantage.

2. A second context for the problem is the emerging field of *mobile multi agent systems*. This promising domain strives for the introduction of global peer to peer computing in a way that offers vast advantages over current day standard client server architectures [CDGM97]. If this really happens, the problem of concurrency interface conflicts will be even more prominent.

3. Following the current trend of embedded systems, we consider a last context that is the field of *connected embedded devices* such as domotic systems, intelligent appliances (ambient intelligence [DBS$^+$01]) and so on, all connected, using, for example, bluetooth. I.e. This is the field of consumer electronics which are mutually connected. Here the problem of interface conflicts is also likely to occur on a much larger scale.

We will now further elaborate on these contexts one by one.

1.1.1 First Position: Standards in Open Distributed Systems

As explained before, the first context for the problem consists of the inherent absence of standards in the world of open distributed systems. The word 'inherent' in the preceding phrase was chosen deliberately. In what follows, we will argue that, in the market driven world in which companies relentlessly compete to make their market position more stable and larger, there cannot be one standard used by everybody. As a consequence companies have a strategic advantage if they can resolve the problem of conflicting standards quickly.

Before 'proving' these statements, let us first explain what we mean exactly by the term 'open distributed system'. A distributed system is a system which consists of two or more computers that are connected to each other by means of a network. Two computers connected with a serial cable in the same room is as much a distributed system as are computers connected via satellites in different parts of the world. Distributed systems can be either open or closed. Open distributed systems are systems made of components that may be obtained from a number of different sources which work together as a single distributed system. These systems are basically "open" in terms of their topology, platform and evolution: they run on networks which are continuously changing and expanding, they are built on top of a heterogeneous platform of hardware and software pieces, and their requirements are continuously evolving [MW99, Kie96, Cro96, CD99]. The Internet is the best example of an open distributed system, because no end user, or company knows the interconnection of all the computers on the Internet. The Internet interconnects different companies as well as end-users by means of an unknown network.

There cannot be *one* standard.

Programming open distributed systems is difficult... not only because the field is relatively new, but mainly because such systems highly depend on many external factors. Not the least important of these is the market behavior of mutually competing middleware providers. Being dependent on their products, one has to anticipate what they are planning to do, and possibly change one's behavior accordingly. If the providers are planning to drop support for a certain standard, one can no longer rely on that standard and might need to look for other solutions. This is nothing new, but for open distributed systems this highly increases the costs of evolution maintenance. Components that exist and are currently widely accepted can be invalid tomorrow. Everything, including the operating environment (that is, the Internet) is so unreliable that writing distributed programs is sometimes said to be nothing more than making some educated guesses about what will happen and what will not happen. [OHE96, Sel01]

The problem of conflicting interfaces cannot be solved by 'requiring' that everybody follows the same standard. Standards evolve not only as a consequence of technological innovation but also as a consequence of the market behavior of the companies behind it. An example of this is the 'evolving' web technology. E.g., whenever companies serve data on the web they are among others dependent on the W3C HTML standard for describing the content of their data. This turns out to be a problem in itself, because this widely used standard is interpreted and augmented in many different ways: Mosaic, Netscape Navigator, Microsoft Internet Explorer, Opera, Konqueror, Mozilla each have their way to understand and render webpages. Surely, one of the reasons behind this diversification comes from the fact that the standard was not very well defined in the eighties. So, the standard needed to grow, which it did. However it did not become a better standard. Instead, different companies tried to redefine the standard to enlarge their market-shares by 'adopting' this standard and modifying it as necessary. One of them was Microsoft. Around 1995 Microsoft was a large player in the operating system market and had the resources to push its own modified standard [BL01] by a) bundling its browser with its operating system, b) not releasing its API's to its competitors Netscape, c) creating components which only worked with its own browser (ASP) and d) creating interpreters that only worked for programs written with its tools (Java versus J++). From these 4 techniques, the last three are centered around the use of standards. Techniques likes this are not exclusively used by Microsoft, which illustrates sufficiently how different standards are defined and molded through the interaction of different companies. Standards are subject to considerations other than the ones involved with the standard. Therefore there cannot be one single standard.

Open Distributed Systems require Interface Adaptors

As explained above, there cannot be one standard, especially not in an open distributed system. Therefore, companies that develop open distributed applications and that involuntarily depend on all kinds of middleware to do so, spend a lot of resources on maintaining their application(s) in a very volatile environment. If only such a company would have the ability to solve the problem of conflicting interfaces in an automatic way, it would have a strategic advantage compared to others. This concludes the first position to illustrate the need for automatic adaptors between conflicting interfaces.

1.1.2 Second Position: Mobile Agent Systems

A number of academic groups try to foresee what the Internet will be like in 10 years. One such research track led to the concept of *mobile multi agents*. Originally these were conceived as intelligent agents [CDGM97] (as defined by Pattie Maes) which would assist the user with some daily tedious tasks such as sorting email in order of interest, looking over his shoulders and assisting where it deems necessary (Microsoft Clippy), bringing people with the same interests together, and performing other intelligent tasks.

As it turned out, to the contrary of what was predicted, intelligent agents currently do not roam the web in search for information, nor do they bring users in contact with interesting opportunities. Writing intelligent agents requires a lot more than writing simple programs, because much of the necessary infrastructure is missing. Such an infrastructure would connect all computers with each other in a non strict hierarchical way. Every single computer would run a *mobile multi agent system*, which would ensure basic operations for the *agents.* Agents would be able to communicate with each other and migrate to other computers as necessary. In such a peer to peer world, computation would emerge from the interaction between the agents, and not only in the agents themselves. The Internet would become a large universal computer [FDF03], where resources are shared and used as needed. Instead of buying software, users would subscribe to software, completely ignoring *where* the software runs. Such applications would consist of a large number of interacting agents, which are not necessarily written by the same company. Users could have their personal intelligent agent on-line, which could schedule meetings with other intelligent agents. Such agents would learn the preferences of their users and search the net for information within that context. They could be able to book airplanes and plan travel-routes. And when the user travels to the other end of the world, the intelligent agent would follow him, together with his preferences. This agent would take care of recreating his virtual environment, wherever the user goes.

However, from a technical point of view, implementing such agents is often still very difficult. One of the reasons is that in general global peer to peer computational networks are extremely volatile. Aside from the problems of finding communication partners (how does a text editor find a suitable printer in a new environment), and the problems of partial failure (what should happen if the agent dies), we have even bigger problems when interfacing the different agents. Let us e.g. assume that a running agent wants to interface with a printer. All printers will provide a different API. Some printers will require one to take a ticket and will call the originator back with a request to send data. Other printers will redirect one to a printer spooler and yet others might require one to lock them first. It is clear that a programmer will never be able to foresee all such printer-interfaces. Even in cases where it *is* possible to foresee and program all currently available interfaces, it is still impossible to support every possible future change. Indeed, by the time one releases an agent, somebody will probably have written another implementation of the interface, or updated an interface to offer a slightly different implementation. This can happen because this agent, with its new implementation of this interface, needs to run on different (possibly optimized) hardware, or because the operating system is different, or simply because the agent is part of another application, and thus developed by other programmers.

In a global peer to peer environment with billions of users, API's change faster than programmers can write. Every extra assumption made by a programmer about an API increases the chance of failure at some time in the future. Hence programmers cannot exhaustively support every possible interface one might encounter in this volatile setting. Instead, a minimal support for some prototypical interfaces will have to be provided under the assumption that the running agent will be able to adapt its interface automatically and dynamically.

The main reason why adaptors are necessary within mobile agent systems is that migration itself leads to situations where the communication partners' implementations are unknown. They can offer the same API, but the implementation itself will give rise to subtle semantic differences that might lead to incorrect behavior. In section 1.3, after presenting the thesis statement, we will discuss in more detail an example of such semantic differences.

1.1.3 Third Position: Interconnected Embedded Systems

As embedded systems become more powerful and new demands are created, the wireless interconnection of different embedded systems is unavoidable. Indeed, it can be expected that, in the near future, technologies such as bluetooth and wireless Ethernet (IEEE 802.11) will lead to an ever further integration of domotics, PDA's, cellular phones and so on. This is referred to as *interconnected embedded systems* [DBS$^+$01]. This evolution will bring about new conflicts since one no longer knows what the neighboring embedded system will be (is it a television or is it another cellular phone we just happened to detect ?). Anticipating all potentially encountered interfaces is simply impossible.

As explained in [BK01] conflicts arise not only because the interfaces are different, but also because the burden of extra non-functional requirements is all too often neglected. Many of the problems involved with embedded systems come from the requirement to do as much as possible with as little resources as possible. This of course depends on the market for which the embedded systems are made. Consumer devices such as remote controls and mobile phones are under much more pressure to reduce the hardware requirements than mid-scale and large-scale embedded systems such as routers and set-top boxes. Requirements such as real-time requirements (the device should be able to guarantee certain deadlines), speed requirements (the device should be able to deliver a certain bandwidth), memory requirements (the device should be able to work with a limited amount of memory), size requirements (the device should be not larger than ...) and other non-functional requirements lead programmers to re-engineer their software, make it smaller and/or more efficient. This often results in behavioral changes within the software. One such a conflict could for instance arise from the modification of counting semaphores to binary semaphores, simply to reduce the memory requirements.

Essentially, the problems are similar to the ones found in mobile multi agent systems. However the difference between interconnected embedded systems and the agent systems discussed above, is that migration from one environment to another is physical migration of the device, instead of migration of the software only, resulting in a very volatile environment in which a developer cannot exhaustively support every possible interface one might encounter.

This wraps up the third motivation that illustrates the need for automatic adaptors between conflicting interfaces. The three contexts outlined above are the three reasons why we investigated the problem of conflicting interfaces.

1.2 Thesis

BEFORE GIVING AN OVERVIEW OF THE CONTENT OF OUR WORK in the following section, let us first shed some light on the methodological side. First we will propose the thesis statement. Second we will clarify our scientific methodology and validation strategy.

1.2.1 Thesis Goal

As explained in section 1.1.1 companies are currently faced with the problem of linking to all kinds of frequently changing third party interfaces and, as seen in section 1.1.2 and 1.1.3, with the advent of open peer to peer networks and interconnected embedded devices, this problem will become even worse because interface conflicts will occur more frequently. Solving such interface conflicts is generally done by inserting adaptors at the appropriate places. However, in very volatile environments creating such adaptors manually & marketing them might be very expensive. Therefore we will investigate how one can create intelligent adaptors that adapt their behavior when new interfaces are encountered. This intelligent adaptor should be able to learn how to resolve interface conflicts in such a way that a) the required behavior of the involved components can be executed over the adapted interconnection and b) the adaptor is able to work on-line in an open system. With on-line we aim at the penalty involved when making an error. In an on-line setting, every wrong decision (that possibly brings an entire application down) is far more catastrophic than the same decision in an off-line setting. This can be because end-users have started the application and are waiting for something to happen, or because a wrong behavior has indirectly an indirect impact on end-users because it introduces and invalid state within the application. It is clear that an on-line setting is far more delicate than an off-line setting.

In the following sections we will explain which methodology we will use to validate this claim and which case we will use throughout the dissertation.

1.2.2 Methodology

Regretfully, a lot of research in the context of open distributed systems and peer to peer networks is based on unrealistic assumptions, such as 'the Internet is fully interconnected' or 'latency will be neglected for the sake of the argument'. During the course of our work we have been constantly taking heed not to make such assumptions. Hence, the methodology used throughout this work is to take into account the boundaries imposed by open distributed systems, such as high latencies, low speeds and unreliable networks. It is within those boundaries that we have tried to create an intelligent interface adaptor.

To construct the intelligent adaptor we tried to use *deterministic reasoning algorithms* as much as possible. However, when these were not available (because of intractability), or inadequate (because of NP-completeness or worse) we used *learning algorithms* with 'less certain' outcome. However, a plethora of learning algorithms exist, each with their own characteristics and prototypical problems and/or solutions. Therefore we had to find out which are suitable ones and which are not. As such, we investigated the use of genetic algorithms, genetic programming, neural networks, reinforcement learning and a lot of variations on them. As we will demonstrate in the dissertation, it turns out that they *can* be useful if used in the correct context and modified accordingly. E.g., in our experiments we noticed that, although some forms of genetic algorithms in theory are perfectly applicable, their practical deployment in the context of open distributed systems was extremely limited because of their off-line nature.

To validate our thesis statement practically, we have selected a particular case for which we show how to create an intelligent interface adaptor, namely an interface adaptors to solve concurrency conflicts. It is in the context outlined above, that we have empirically validated which algorithms were practically feasible, based on certain requirements, in order to overcome unanticipated concurrency conflicts of two or more communicating components.

1.2.3 Case & Thesis Statement: Concurrency Conflicts

When two or more components interact concurrently, essentially two kinds of problems can occur which we might call *functional* conflicts and *non-functional* conflicts. Functional conflicts emerge when the interacting components are speaking a different language with regard to their functionality. E.g. one component might be 'about' booking flights and another might be 'about' reserving library books. Clearly the interfaces of these components are completely incompatible and it is not the goal of our work to do something about it. Non-functional conflicts emerge when the components *are* essentially speaking about the same thing, but in a different way. Examples of non-functional problems are concurrency problems such as race-conditions, deadlocks, livelocks, starvation and others. These typically occur when the order in the message interaction is wrong. To solve these problems every participating party implements a so-called concurrency strategy. However, when different concurrency strategies do not cooperate in the right way, a conflict arises. Having stated this, we can now repeat the thesis statement given in section 1.2.1 in its full context.

It is possible to create, for certain categories of concurrency interface conflicts, a concurrency-adaptor that learns how to resolve the conflict in such a way that a) the required behavior of the involved components can be executed over the adapted interconnection and b) it is able to work on-line in certain categories of open system. We validate this by constructing such a concurrency-adaptor.

1.3 A Preliminary Example

In this section we introduce a preliminary example of the problem we will investigate. In particular this example concerns a concurrency conflict between a client and a server. After presenting the example, we will discuss what we consider to be a conflict and what requirements we pose upon an adaptor. Throughout this section we will give forward references to those chapters that discus the matter in more detail.

1.3.1 What is an Interface ?

For clarity, we assume that the only way through which the state and/or behavior of a component can be altered is through one of its access points. Such an access point (or group of access points) will be called an *interface*. (See chapter 2 on page 25). Below we describe the interfaces of our two example components by means of incoming and outgoing messages with some extra non-formal documentation.

The Interface of the Server Component

The server is a whiteboard that allows multiple clients to lock certain position before they can act upon these position.

The server's API:

```
incoming lock(x, y)
outgoing lock_true(x, y)
outgoing lock_false(x, y)
  // lock_true or lock_false are sent back whenever a lock
  // request comes in: lock_true when the resource is locked,
  // lock_false when the resource couldn't be locked.
incoming unlock(x, y)
outgoing unlock_done(x, y)
  // will unlock the resource. Send unlock_done back when done.
```

The server component also offers some behavior on another port. This behavior is as simple as possible.

```
incoming act(x, y)
outgoing act_done(x, y)
  // will perform action on the component
```

The Interface of the Client Component

The API of the client component:

```
outgoing lock(x, y)
incoming lock(x, y, result)
  // lock(*, *, true) or lock(*, *, false) are received whenever a
  // lock request comes in. Before performing an act operation the
  // server will be locked by sending out a lock.
outgoing unlock(x, y)
  // will unlock the resource. No message is returned.
  // This operation always succeeds.
```

The client component expects some behavior from the server

```
outgoing act(x, y)
incoming act_done(x, y)
  // will perform some action on the server
```

Both interface descriptions should be read as messages that are sent and received at runtime, without prior knowledge of the communication partners. This means that any interface conflicts between those two interfaces cannot be checked at compile-time. They can only be detected when the components execute.

1.3.2 A Syntactical Conflict

If we look at both interfaces, we see immediately 2 essential differences between the client and the server. First, whenever the client requests a lock there will occur a *syntactical conflict:* the client expects the server to return a `lock(x, y,result)` message, while the server will return a `lock_true` or `lock_false` message. A second similar conflict can be found when looking at the `unlock` operation. The client will only send out an `unlock` request, while the server will send back an `unlock_done` message. This message will not be understood.

To solve this problem one can easily insert an adaptor between both communicating components. This adaptor will offer two interfaces: one interface aimed at the server side, another interface aimed at the client side:

```
incoming lock(x, y)
outgoing lock(x, y)
  // when a client requests a lock, this lock will be sent
  // through to the server
incoming lock_true(x, y)
incoming lock_false(x, y)
outgoing lock(x, y, result)
  // when the server responds with lock_true this request will
  // be translated towards the client as lock(x, y,true).
  // When the server response with lock_false this request will
  // be translated towards the client as lock(x, y,false).
incoming unlock(x, y)
outgoing unlock(x, y)
incoming unlock_done(x, y)
  // when a client requests an unlock, this message will be
  // forwarded to the server when an unlock_done() message
  // arrives from the server this message will be silently ignored
incoming act(x, y)
outgoing act(x, y)
incoming act_done(x, y)
outgoing act_done(x, y)
  // similarly, the action request will be translated by simply
  // passing any act or act_done message.
```

1.3.3 Conflicting Semantics

Aside from the syntactical conflicts between both interfaces, that in this case, can be easy mediated, semantic differences can also occur. The documentation does not state how the components implement their locking strategy. For instance, one can implement a counting semaphore or a binary semaphore as a concurrency strategy.

- **Counting Semaphores:** allow a client to lock a resource multiple times. Every time the resource is locked the lock counter is increased. If the resource is unlocked the lock counter is decreased. The resource is finally unlocked when this counter reaches zero. These semantics allow us to use routines which autonomously lock resources.
- **Binary Semaphores:** provide a locking semantics that doesn't offer a counter. It simply remembers who has locked a resource and doesn't allow a second lock. When unlocked, the resource becomes available again.

Differences in how the programmer considers the *lock* and *unlock* operations can give rise to another branch of interface conflicts. This is illustrated in figure 1.1. If the client agent expects a counting semaphore from the server agent, but the server agent offers a binary semaphore, then, the client can lock a resource twice and expects that the resource can be unlocked twice. In practice the server just has marked the resource as *locked*. If the client now unlocks the resource, the resource will be unlocked. Acting upon the server now is impossible, while the client expects it to be possible.

1.3.4 What is a Conflict ?

In the above example we gave an example of a conflict between a counting semaphore and a binary semaphore. However, this conflict only pops up when the client makes use of certain possibilities hidden within the semantics. Specifically, the client needs to decrease a lock-counter, without bringing it to zero and then act upon the supposedly locked resource. A behavior such as this is not necessarily present within the client component. A client component might also simply lock a resource multiple times and then unlock it again until the lock-counter reaches zero. If the client behaves as this, we can barely say that there is a conflict between the two interfaces (aside from the obvious syntactical conflict).

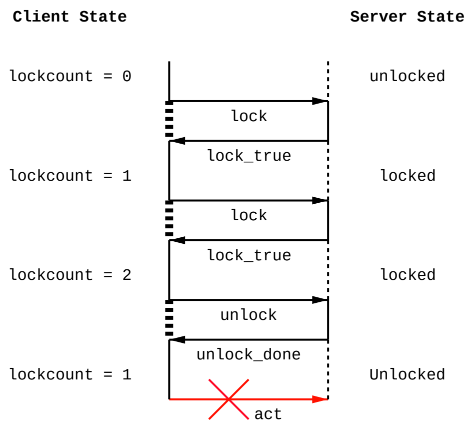


Figure 1.1: An interface conflict when the client agent expects a counting semaphore from the server agent and the server agent only offers a binary semaphore.

In other words, depending on the *behavior* of the involved components, a possible conflict might be a real conflict or not. This leads us to define an interface conflict in terms of the required behavior:

If the overall required behavior of a set of components cannot be executed by only using the interfaces between the different components then the interfaces of these components are in conflict.

This definition should be clarified to a certain extent. We mention 'the overall required' behavior. This implicitly means that we assume that all the involved components have agreed to follow a certain behavior. For instance, if all component agree to *not* follow a concurrency strategy then we cannot say that the previously mentioned interfaces are in conflict. If both components have agreed to follow a locking strategy then we can declare the interfaces to be in conflict. However, if not all components agree to follow a certain behavior then this definition of interface-conflicts is useless because it will be virtually impossible to resolve the conflict. E.g.: if the server requires a concurrency strategy but the client does not agree to follow any concurrency strategy at all, then the notion of an interface-conflict is useless because it cannot be measured against a required overall behavior.

In this dissertation we will assume that the involved components have already agreed to provide/require a similar behavior. In chapter To better understand the kind of conflicts we are dealing with we will discuss different concurrency strategies and define an implicitly agreed overall behavior in chapter 5 on page 79. After doing so, we will investigate in chapter 6 on page 101 the conflicts that can arise in such a situation.

1.3.5 What Are the Requirements ?

To verify whether it is possible to generate an intelligent adaptor we need to specify the requirements for such an adaptor. Below we present the requirements which an adaptor should satisfy. In chapter 7 on page 123 we will do this in detail for the concurrency-adaptor.

Given the definition of *interface conflicts*, together with the notion of *overall required behavior* we can easily understand that an adaptor works when it is

1. able to mediate the communication between the involved components in such a way that the required overall behavior can be executed.

2. The adaptor itself should be able to work at *runtime* and

3. it should work by *only* modifying the message flow between the involved components.

If there is no full agreement on the required behavior then the adaptor should try as hard as possible to satisfy as much as possible of the required behavior. For instance, if one of the clients doesn't follow any concurrency strategy at all, then the adaptor should allow this client to do whatever it wants, without interrupting the behavior of the other clients that follows a concurrency strategy. Similarly, when all clients expect a concurrency strategy and the server doesn't specify one, then the adaptor should try to coordinate the different behaviors in such a way that everyone's behavior is present. In particular it is useful to note that certain concurrency strategies might lead to deadlocks. In this case the resulting adaptor works correctly if the resulting behavior also leads to deadlocks in the same situations. This is consistent with the requirement because in both examples *all* required behavior can be satisfied. In situations where the overall required behavior is inconsistent, thus where it can never be completely satisfied, we will refrain from verifying the working of the adaptor.

The reason why we require the adaptor to work at runtime is mainly because this embodies a more realistic approach. In open distributed systems, one component might suddenly need to talk to a previously unknown component. If the adaptor is able to work and mediate conflicts at runtime, then it will also be able to learn how to behave when such a new communication partner is encountered in an open system. however, if the adaptor is only able to learn a suitable behavior off-line, then it might not be possible to adapt correctly in a running open system.

The reason why we only want to modify the message flow between the components is because in an open system it might not always be possible to modify the source of the involved components. Therefore, the only thing that such an adaptor can be allowed is modifying the message flow between the different components.

The high-level definition of interfaces, conflicts and the requirements given in this section can also be found in [VHT00].

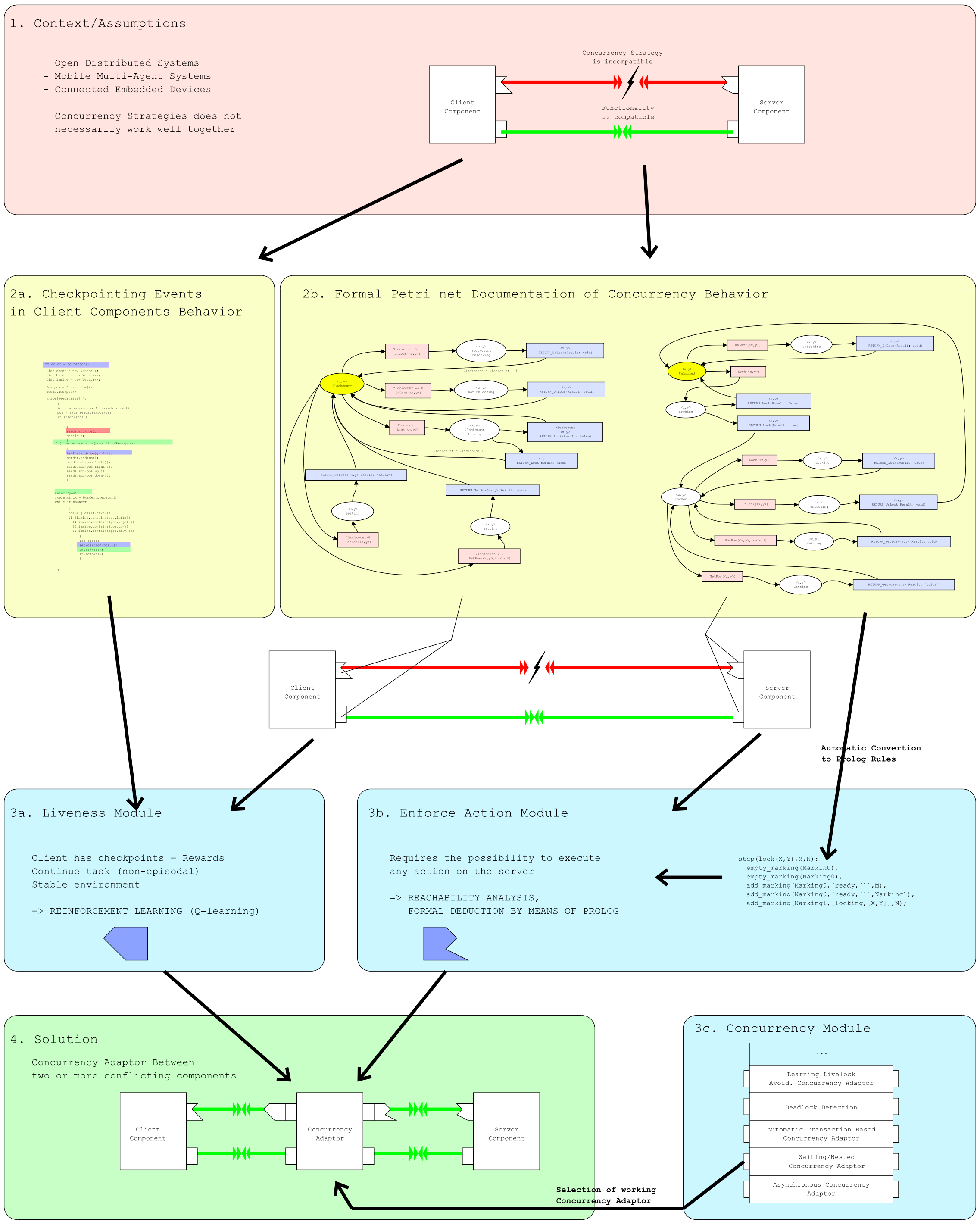
1.4 Structure of Our Work

As we will demonstrate in the dissertation, the technical characteristics of the reasoning algorithms and the learning algorithms we experimented with, lead us to formulate every adaptor as a suite of three cooperating modules. The basic idea is that a concurrency conflict interface adaptor will consist of one 'central' module that contains the actual solution for the concurrency conflict; together with two peripheral modules whose task is to mediate between a component and the central module in some way. In the following sections, will elaborate on the requirements we impose on the developer of the interacting components in order to facilitate the construction of the mediating modules. Furthermore we will shed some light on their functionality and on the reasoning and learning algorithms we adopted in order to construct them.

1.4.1 Adaptors

We will now briefly sketch how an intelligent adaptor can be constructed. This adaptor will be placed centrally between all communicating components. To a certain extent this is contradictory to the motivation of open distributed systems. However, without the ability to intercept all communication it is often impossible to solve concurrency conflicts between multiple partners. We will come back on this issue in section 12.7.1 on page 197 and in section 6.4 on page 113 where we will discuss possible solutions to this.

As explained, the adaptor will learn how to mediate conflicting concurrency strategies between running components. Typically this will be achieved by trying out different actions within different situations and learning which action is appropriate at what moment in the message sequence of those components. However, this learning-phase also happens at runtime, so the adaptor must be sure that no chosen action interferes with the correct working of the application.


1. Context/Assumptions
- Open Distributed Systems
- Mobile Multi-Agent Systems
- Connected Embedded Devices
- Concurrency Strategies does not necessarily work well together
Concurrency Strategy is incompatible
Client Component
Functionality is compatible
Server Component
2a. Checkpointing Events in Client Components Behavior
2b. Formal Petri-net Documentation of Concurrency Behavior
Client Component
Server Component
Automatic Convertion to Prolog Rules
3a. Liveness Module
Client has checkpoints = Rewards
Continue task (non-episodal)
Stable environment
=> REINFORCEMENT LEARNING (Q-learning)
3b. Enforce-Action Module
Requires the possibility to execute any action on the server
=> REACHABILITY ANALYSIS, FORMAL DEDUCTION BY MEANS OF PROLOG
4. Solution
Concurrency Adaptor Between two or more conflicting components
Client Component
Concurrency Adaptor
Server Component
Selection of working Concurrency Adaptor
3c. Concurrency Module
...
Learning Livelock Avoid. Concurrency Adaptor
Deadlock Detection
Automatic Transaction Based Concurrency Adaptor
Waiting/Nested Concurrency Adaptor
Asynchronous Concurrency Adaptor

```
step(lock(X,Y),M,N):-
  empty_marking(Markin0),
  empty_marking(Narking0),
  add_marking(Marking0,[ready,[]],M),
  add_marking(Narking0,[ready,[]],Narking1),
  add_marking(Narking1,[locking,[X,Y]],N);
```

Figure 1.2: Overview of the work.

This forms a general problem because an adaptor does not have this knowledge available. Therefore we will require that all components offer a full formal documentation of their interfaces. This formal documentation will be under the form of colored Petri-nets and statically placed checkpoints within the source code (the yellow boxes 2a and 2b in figure 1.2). We will further elaborate on the documentation in section 1.4.5. With the availability of this documentation, an adaptor can readily experiment in a runtime system with the assurance that no chosen action will corrupt one of the running components.

As already stated, the adaptor itself (depicted as the central white box in the green box in figure 1.2) is a chain of three smaller connected modules (the blue boxes 3a, 3b and 3c in figure 1.2), each responsible for a different role in adapting the concurrency conflicts that might exist when one component tries to communicate with another through non-compatible interfaces. A component that initiates a communication uses a certain *required* concurrency strategy in order to communicate with the second component, which offers a *provided* concurrency strategy. In general the idea is that the different concurrency strategies will be converted to an intermediate protocol which will be used to order incoming *functional* requests.

1. At each step in time, the required concurrency strategy expects the provided one to be in a certain state. It is the task of the *enforce-action module* (box 3b) to convert an intermediate protocol into an effective realization of this state. In other words, the enforce-action module assures that an expected state of the required concurrency strategy can be enforced upon the provided concurrency strategy.

2. The *liveness module* (box 3a) converts a required concurrency strategy to an intermediate protocol, suitable for a general concurrency module. Its goal is to keep the underlying client component alive by returning a correct concurrency strategy.

3. The central *concurrency module* (box 3c), is placed in between the two others, and understands the intermediate protocol of both modules. This module is the actual concurrency strategy implemented. This module honors certain important criteria such as no-races, liveness and others... Hence, the concurrency module receives a certain set of message from the initiator, that *should* be executed and will automatically insert appropriate concurrency messages to execute the required sequence.

We will now elaborate further on every module in the following three sections.

1.4.2 The Enforce-Action Module

The enforce-action module (blue box 3b in figure 1.2) will automatically *bypass* the provided concurrency strategy. This is done by analyzing the possible future traces and finding out how certain actions (or states) can be *enforced* upon the component. The formal analysis we use to bypass the concurrency strategy is done by means of a prolog program and requires a formal documentation of the provided concurrency strategy. (yellow box 2b in figure 1.2). Bypassing a concurrency strategy is of course a dangerous thing to do because race conditions can happen from then on. To avoid these race conditions, all communication with the bypassed component should henceforth go through the concurrency adaptor. In chapter 8 we explain how we do this.

1.4.3 The Liveness Module

The liveness module (blue box 3a in figure 1.2) will automatically *learn* what kind of messages the required concurrency strategy would like to receive back at a certain moment during the communication. To illustrate this, one can easily see that a component would like to receive back a `LockTrue` message when it requests a lock. A `LockFalse` would also be possible of course, but it is clear that in this case, this is not the answer that is favored by the component. As we will explain in chapter 9 learning this required behavior is done by means of *reinforcement learning* (Q-learning more specifically). The answer favored by the component is used to create a suitable reward/punishment for this learning algorithm. Of course, the algorithm itself cannot determine the favored answer such that we will have to rely on the developer to specify this. As we will show in section 9.2, this is done by means of *checkpointing* the source (yellow box 2a in figure 1.2).

1.4.4 The Concurrency Module

The concurrency module (blue box 3c in figure 1.2) is placed in between the liveness module and the enforce-action module (hence, between the required and provided concurrency strategies). As explained before, this central module contains the 'actual' concurrency strategy used between the components adapted by the other modules. The specific concurrency strategy plugged into this module is actually outside the scope of our work: it can be any strategy of choice which works well in the particular environment in which our techniques will be deployed in. These contents of the module can thus be taken from a repository of 'good working concurrency strategies' or can be a learning concurrency strategy that learns how to optimize the locking strategy to avoid rollbacks and livelocks. In chapter 10 we will elaborate further on the possibilities of filling up this module.

1.4.5 Formal Documentation

In section 1.4.1 we explained that the reasoning and learning algorithms we used to make a working interface adaptor, are based on formal documentation that has to be provided by the programmer of the communicating components. More concretely, in the realization of the above three modules, we relied on extra input, supplied by the programmer of the required and provided concurrency strategies: he has to specify when his component is 'alive' and he has to document the interface in a formal way. The formal documentation we use are so-called colored petri-nets [Jen94, Lak94, KCJ98, EK98]. This formalism is suited for specifying which actions are enabled at some moment in time, and, has some additional properties that are both highly favored by human readers, and are well-suited for automatic interpretation by algorithms:

1. They are specified by means of a *graphical representation*. A representation that is intuitive and covers in one drawing enough detail to understand what the represented model is about.

2. Petri-nets have a description of both *states and actions*, this in contrast to state diagrams or transition diagrams, which cover only part of the behavior of a system.

3. Petri-nets are a formalism that can describe a system at *any level of abstraction*. Petri-nets can be used to describe the interaction between high level modules as well as the full interaction within these modules. Petri-nets can be described to specify a large variety of different systems.

4. *Petri-nets are stable* with respect to minor changes of the modeled system. It means that small modifications of the modeled system do not require a complete rewrite of the Petri-net. In many other description languages this is not the case (e.g.: finite automaton).

5. Petri-nets can be analyzed through a large number of *formal techniques*, by which properties of the modeled system can be verified. This includes: construction of *occurency graphs* (to determine which global states are reachable), calculation of *invariants* (pre- and post- conditions checking), *reductions* (shrink down a Petri-net but still preserve a number of properties) and checking of *structural properties* (such as starvation or deadlocks). From these, we need reachability analysis and checking of deadlocks.

Aside from these nice properties, we think there are even more reasons why programmers ought to use them more often in order to document components formally. Here are some additional advantages of using such a formal description:

1. A first advantage is the possibility to verify automatically whether a component adheres to its specification. With such a check one can verify that the component does not send out certain messages when they are supposed not to be sent out.

2. A second advantage is the possibility to test the robustness of a component automatically. By using the formal interface description, a testing adaptor can choose which message it sends back in a certain context. Not necessarily all messages will be understood by the program in a given context. By testing a component this way we assure that components are robust towards their own specification.

These points are very important because they make it possible to automatically check the implementation with the specification and vice versa. This leads to better communication of the behavior of a program towards other developers. However, as we show in our work, it will also provide us with a way to communicate the behavior of the program towards another program, namely the algorithm that is responsible for determining the interface adaptor and resolving the concurrency conflict. We will go deeper into the role of Petri-nets for this in chapter 3.

1.5 Scientific Approach

IN CONTRAST TO PHYSICS, CHEMISTRY AND OTHER DISCIPLINES, computer science often lacks an objective and fixed scientific method. The non formal characteristics of many problems often leads to a lack of investigation depth and rigor. Furthermore, given the current possibilities of computers, it is very easy to create layers of abstraction upon layers of abstraction, often without having the scientific yardsticks to verify their value and how they contribute to the progress of the field. According to [MSBW94] there are three possible things one can contribute in the field of experimental computer science.

- ⇝ A proof of existence, whether a certain problem can be solved and how it can be solved.
- ⇝ A proof of concept, which merely illustrates that a certain concept is useful in certain environments.
- ⇝ A proof of performance, which in general proves that a new algorithm works better than another algorithm.

The work presented in this dissertation is essentially a proof of existence. We demonstrate that we *can* write a concurrency adaptor between different conflicting concurrency strategies. We demonstrate this in a *constructive* way by creating a set of steps that will lead to such a non-trivial, correctly working adaptor. One of the side-contributions of this dissertation is a proof of concept: by using the technique of Petri-nets to document interfaces we have clearly demonstrated its usability for it allows automatic black-box testing of components as well as the ability to automatically compare the documentation with the implementation.

This dissertation makes no claim whatsoever about performance. Solely for the interested reader some performance estimates are given, but these should be read as purely informative.

1.5.1 Publications

The research presented in this dissertation, as well as the research that leads to this dissertation has been reported on internationally:

1. Werner Van Belle, Tom Mens, Theo D'Hondt
   Using Genetic Programming to Generate Protocol Adaptors for Interprocess Communication
   Published in Evolvable Systems: From Biology to Hardware, Proceedings of the 5th International Conference on Evolvable Systems (ICES2003)
   Editors: Andy M. Tyrrel, Pauline C. Haddow, Jim Torresen

Lecture Notes Computer Science 2606
Pages: 422 - 433
Springer Verlag; Mars 2003

2. Stefan Van Baelen, David Urting, Werner Van Belle, Viviane Jonckers, Tom Holvoet, Yolande Berbers and Karel De Vlaminck
Toward a unified terminology for component-based development
ECOOP2000 workshop on Component-Oriented Programming (WCOP)
June 2000

3. Werner Van Belle, Johan Fabry, Theo D'Hondt and Karsten Verelst
Experiences in Mobile Computing: The CBorg Mobile Multi-Agent System
Proceedings TOOLS Europe 2001, volume 38, pages 1-9
Editor: Wolfgang Pree
IEEE Computer Society Press; Zurich, March 2001,
http://borg.rave.org/

4. Werner Van Belle, Karsten Verelst and Theo D'Hondt
Location Transparent Routing in Mobile Agent System – Merging Name Lookups with Routing,
Proceedings Workshop on Future Trends of Distributed Computing Systems, volume 7, pages 207–212,
IEEE Computer Society Press, December 1999,
http://borg.rave.org/

5. Werner Van Belle and Theo D'Hondt
Agent Mobility and Reification of Computational State, an experiment in migration,
Published in International. Proceedings of Infrastructures for Agents, Multi-Agent Systems and Scalable Multi-Agent Systems
Editors: Tom Wagner and Omer Rana
Lecture Notes in Artificial Intelligence (LNAI)
Pages: 166 - 173
Springer Verlag; June 2000

1.6 Related Work

THE MOTIVATION THAT LEAD TO THIS RESEARCH has also motivated other researchers to investigate similar problems. In this section we discuss similar approaches such as adaptor generation by means of state machines and adaptor repositories. Afterwards we discuss alternative approaches such as languages for writing adaptors, fundamental problems of open protocols, ontologies and others. Related work that is technically relevant will be discussed when appropriate. Section 3.3.1 on page 41 discuses alternative approaches to Petri-nets. With respect to the problem of conflicting concurrency strategies there is, to the best of my knowledge, nothing to be found.

1.6.1 Similar Approaches

Adaptor Generation by Means of Finite Automata

In this approach an adaptor is considered to be a program that is generated when a description of the different interfaces is known. To make the required behavior executable over the conflicting interfaces, a lot of research focuses around the generation of finite automata. The work of [Wyd01,Reu] covers how such an automata-adaptor can be generated. The work itself is aimed at the composition of software components in closed systems. The strategy used relies on the availability of place-holders that abstractly describe what kind of interaction is required, while an finite state machine (written down as an MSC) describes the behavior of the interfaces themselves. In our work we have actively sought for an example where such place-holders cannot be written easily, i.e., where the requirements cannot easily be expressed as a finite state machine.

We believe concurrency is such an example because a requirement such as 'no race conditions should occur' is difficult to write down as a finite state machine. The work of [Wyd01] also relies on the availability of a learning algorithm, however not much detail has been given on this algorithm. This kind of approach also has the problem of 'non-symbolic operation', which means that the finite state machines cannot express actions on arguments, such as 'add those two arguments together and pass it through'. In short, finite state machines are limited in their expressive power.

Software Integration: Adaptor Repositories

Another approach relies on a repository of adaptors that ought to work in a certain application domain. Creating such adaptors is done off-line whenever a conflict arises. For every occurring conflict one can develop an adaptor manually. The strength of this approach is that a) a better quality control can be enforced upon the entire adaptation process, b) there virtually no limitations are on the requirements posed upon the adaptors because they are manually written and c) in the name of efficiency, only real occurring conflicts will be solved. The biggest limitations of such techniques are their scalability. On one side, for every new component there exist potentially as many conflicts as there exist other components. This means that the number of adaptors might be quadratic to the number of components. By defining a common ground on which adaptors can be written, thus by exploiting domain specific features, one might be able to reduce the problems of managing this growth.

The biggest reason why we didn't investigate this track further is because it is more a *process* towards a solution than a solution in itself. Nowadays, companies such as IBM are selling 'integration' and are efficiently creating adaptors by relying on domain specific features.

1.6.2 Alternative Approaches

In this section we cover research that uses another approach than the one we have created.

Undecidability & Possibilities

The idea of creating self-adapting protocols is certainly not new. Open protocols are protocols which optimize the communication themselves by modifying the protocol they are using at runtime. However, research done by Vreeswijk [Vre95] shows that in general it is impossible to create a fully open protocol. Certain restrictions will always be in place. In our dissertation this is also the case. We had to make assumptions that limit the applicability of our work to the field of concurrency problems and open distributed systems. Other research approaches the problem of conflicting interfaces more philosophically by testing the boundaries and possibilities offered by open protocols. The Talking Heads experiments is one such an experiment. Here robots learns the 'meaning' of words by communicating about similar objects [SKML].

Version Control and Software Evolution

An alternative approach towards interface conflicts is to avoid them. By carefully tagging different components with a version-number one might be able to reduce a large number of unanticipated conflicts. Instead of using any component available to offer a certain functionality, it becomes possible to select the correct version of a component. However, this does not solve the problem, it only manages it better. After a while, different versions will be working together, thus not necessarily avoiding conflicts. Software evolution research has pointed out that seemingly harmless upgrades to software implementations may result in unanticipated behavior within other parts of the system [LSMD96]. An explicit example of this is given in this dissertation in section 6.1.1 on page 101.

With respect to versioning, [WMC01] discusses a number of different tools and approaches toward the software versioning problem. The paper itself present a unified approach towards versioning. [Sug98] discusses what kind of extensions should be added to .DLL's to make runtime upgrades possible.

A Common Language: Ontologies

From a practical point of view, ontologies [Gru93] try to describe information contained within a system. The representations of this information can be different from system to system. Some research groups investigate how this information can be exchanged between different systems. One such an example is the 'knowledge interchange format', or KIF [GF93]. Nowadays, KIF is being mapped onto XML [ES] such that the use of a DTD can help in interchanging data in real-world environments. However, this kind of approach typically does not allow for *active* processes. [CP95] covers an agent based ontology approach to integrate different applications with each other. [NU97] argues that the availability of a communication channel over which agents communicate over the language they are speaking is essential together with the availability of a shared ontology.

Adaptor Writing

Another approach to the problem of conflicting interfaces is the creation of a language in which one can easily express how the adaptor should work.This approach is taken by Picolla [ALSN01], CSP [Hoa85], which is specifically suited for concurrency. Other approaches handle the problem of conflicting interfaces by investigating how interface conflicts can be avoided by offering a type system that allows for 'open' communication [GK99, JB99] and others. [JB99] also explains why standard object oriented language constructs do not easily allow for easy adaptor creation. KQML [FFMM94] is a language that is designed to allow agents to share and communicate information with each other. [CFL+99] presents a system that uses KQML and Java as an underlying language to write adaptors between conflicting components within enterprises. [BBT01, BBC] discusses a language to coordinate the interaction between different components. Its language is mainly based upon CSP.

In this dissertation we have investigated in what kind of language our adaptor should be written. However, instead of creating (or using) a human accessible language, we have been looking for a language that is better suited for automatic generation by means of computers.

1.7 Roadmap

1.7.1 Chronology

THE ORIGINAL GOAL OF THIS DISSERTATION, which is to create an intelligent adaptor between conflicting interfaces, represents a very difficult problem. Therefore we have tried to get a grip on it by choosing appropriate techniques and identifying important subproblems. Once a problem was identified we have restricted ourselves to it. During our work we have always tried to minimize the number of restrictions and have tried to keep the solutions as general as possible. However, before we were able to create a concurrency adaptor we have tried a number of different paths to tackle the problems involved. The path we have followed is as interesting as the solution itself. Therefore we have tried to retain as much information as possible, such that, if an intelligent adaptor needs to be developed for other domains, this dissertation can be used as a guide. To help the reader in understanding the structure of this work, we will now summarize the path we have taken. Note that this path is in chronological order and not in the order of chapters.

1. *Definition of the problem and the environment*. The first thing we have done was defining the problem and creating an environment in which we could perform our experiments. In our case the problem were conflicting concurrency interfaces. In chapter 1 we have explained

why we need intelligent concurrency adaptors. Before we could experiment with conflicting concurrency strategies we needed a realistic model of the environment in which these conflicts would arise and in which adaptors could be written easily. Therefore, we have modeled our problem in a system which hides much of the technical problems programmers face with current day architectures. The system itself is event based and correctly represents open distributed systems. In chapter 2 we present this event based system, together with examples of how components and adaptors can be written.

2. *Explore the domain*: Secondly, we have explored the domain of conflicting concurrency interfaces. To do this, we have created a number of different concurrency strategies, based on real-world examples. Chapter 5 introduces these and investigates the differences between the different concurrency strategies. Based on these differences, which we call variabilities, we have made a list of conflicts. In chapter 6 we cover all the conflicts we have investigated, together with a discussion how we could solve every conflict. Part of exploring the domain consisted of a small experiment in which we tried to generate an adaptor *fully* automatically. From this experiment we learned that if we want to mediate differences between conflicting interfaces, the adaptor needs more information than is commonly found in interface descriptions.

3. *Which information is needed*: After the small tests, we observed the need to specify interfaces in a formal way, such that much more information is available for creating an adaptor. In chapter 3 we explain how Petri-nets can be used to describe interfaces.

4. *Preliminary experiments*: With this extra information, we tried to generate an adaptor automatically, without exactly knowing what the requirements were. From these experiments (which are detailed in chapter 11), we learned which requirements were necessary.

5. *Specify the requirements*: After preliminary tests, we were able to specify the requirements of a concurrency adaptor. The requirements we will present are chosen in such a way that, if they are satisfied, cover most of the problems investigated in the previous phase. Chapter 7 describes what exact requirements we needed for an intelligent concurrency adaptor. A second result from the preliminary tests was that it became clear that we would have to modularize the adaptor because we started to understand the limitations of the techniques we were applying.

6. *Modularization of the adaptor*: To be able to meet all our requirements, we needed to modularize the adaptor. Different parts of the adaptor were involved with different functionalities to meet different requirements. In chapter 7 we describe (aside from the requirements) also the way we have modularized the adaptor. The result consists of three modules. The first module, an enforce-action module will mediate the conflict between the adaptor and the server. Chapter 8 explains how the *enforce-action* module bypasses a provided concurrency strategy. To do so it will make use of a *prolog* program to automatically deduce how to reach a certain state within the server. The second module, a liveness module mediates the differences between the adaptor and the client and will decide what to do in certain situations. This module makes use of a learning algorithm to offer correct feedback behavior toward the clients. To explain this we have covered in chapter 4 a number of learning algorithms (reinforcement learning, genetic algorithms and classifier systems). The liveness module itself is documented in chapter 9. The last module, the concurrency module, is the actual adaptor, which will decide what to do, given a certain situation. This module is documented in detail in chapter 10. We explain how a concurrency strategy can be inserted in this module such that the previously stated requirements are met.

7. *Experiments & Validation:* To validate our solution, we have performed tests on all three modules. The results of these experiments have been covered in the different module chapters (8, 9 and 10). Chapter 12 refers back to the concurrency conflicts discussed in chapter 6 and verifies how these are solved by the three different modules. Chapter 13 wraps up the

thesis by recalling the introduction and stating that we have shown that implementing an intelligent concurrency adaptor is possible.

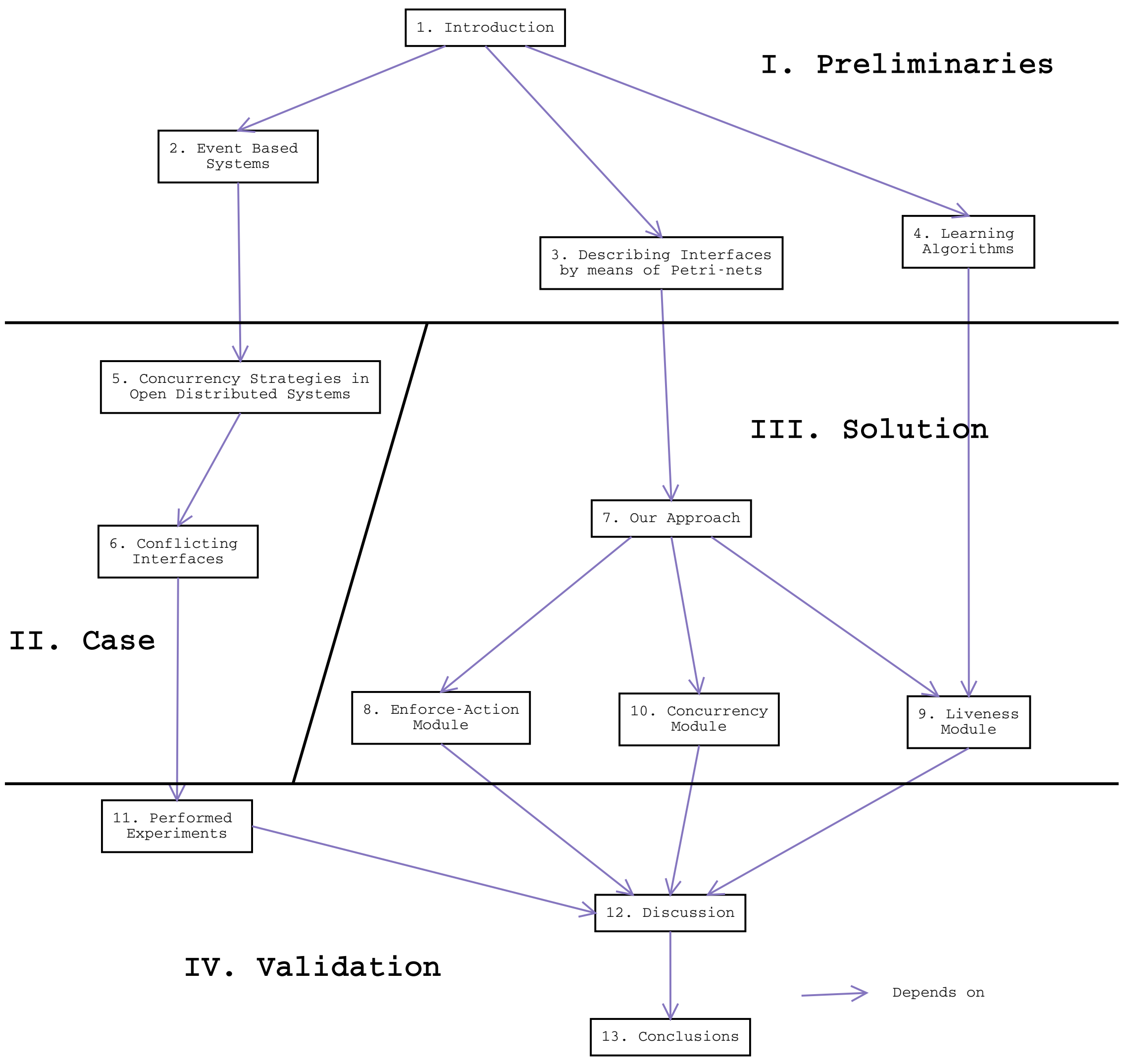


Figure 1.3: Dependencies between chapters

1.7.2 Storyline

However, the presentation of this research has been molded into a standard format, resulting in a text which is divided into four parts: I) preliminaries, II) the case, III) our approach and IV) validation. Part I, the preliminaries covers the event based model we use (chapter 2), explains how Petri-nets can be used to describe interfaces (chapter 3) and introduces the learning algorithms we need later on (chapter 4). The presentation of our cases (part II) covers two chapters. Chapter 5 explains which concurrency strategies we investigate, chapter 6 contains the conflicts we use. The presentation of our approach (part III) starts with presenting a general overview of the solution in chapter 7, after which the different modules are presented in chapters 8, 9 and 10. The last part of this dissertation (part IV), validates our initial claims. We will not only verify the working of our adaptor in chapter 12 and conclude our thesis in chapter 13, but we will also give an overview of the preliminary experiments that have lead to the current setup of the adaptor (chapter 11). Figure 1.3 shows the dependencies between the different chapters.

1.7.3 Conventions

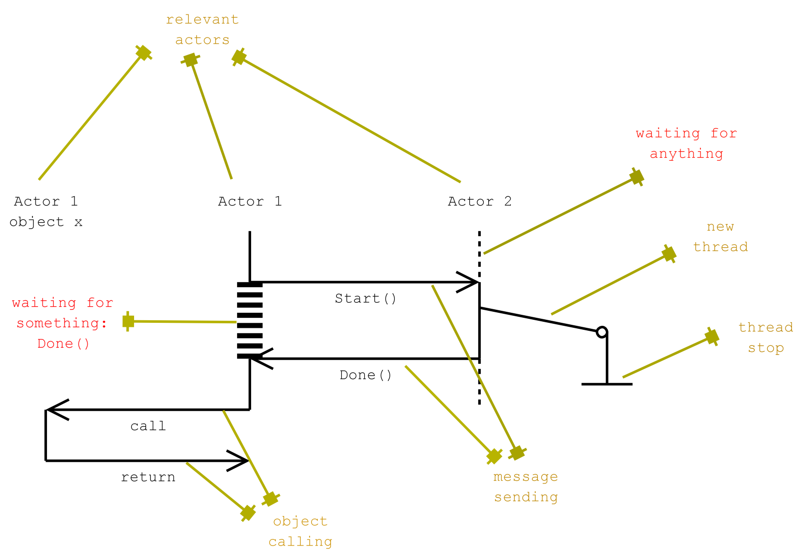


Figure 1.4: Message Sequence charts used throughout the thesis

Throughout the thesis we will make use of a sort of message sequence diagram. The message diagrams are loosely based upon UML, with the difference that we have added some notation to support different threads and better illustrate concurrency problems. Figure 1.4 contains a message sequence chart in which we explain the different notations used:

1. The top line of each message diagram contains the names of the relevant actors. For example, the names of the processes involved, the names of the classes and instances that are relevant for the problem at hand.
2. Below every actor is a vertical line, which represents a thread. The top of the line is time 0. This line is drawn differently, depending on the situation.
   (a) If no execution stack is present, no line is drawn.
   (b) If an execution stack is present and executing, a full line is drawn.
   (c) If an execution stack is present but not executing because it is waiting for *any* thing to happen, a dotted line is drawn. This means that virtually *anybody* can re-initiate the execution.
   (d) If an execution stack is present but not executing because it is waiting for *some specific* thing to happen, a vertical line with horizontal dashes is drawn. A specific thing can be, for example, waiting for a return of a message.
3. If the control flow jumps from actor to actor, we draw horizontal lines (or slanted lines to illustrate network delays). On the line we put a description of the message.
4. Creating new processes is done by splitting an existing process. Here we draw a horizontal line but keep on executing the first process.

5. A control flow ends with a horizontal line.

Figure 1.4 illustrates this convention. Note that the difference between a *call* and a *message* depends on how the control flow behaves. If the thread moves from one actor to another and returns we call it a call. On the other hand, if a thread continues (for instance with actively waiting) after informing contacting another thread, we call it a message send.

We will now start with the preliminaries, in which we present the event based model we use as a basis. Afterwards, we present how Petri-nets can be used to describe interfaces and finally we wrap up the preliminaries with a short introduction to learning algorithms.

Part I

Preliminaries

Chapter 2

Event Based Models for Distributed Systems

SINCE THE GOAL OF THIS DISSERTATION is to write an intelligent adaptor between conflicting interfaces, we need to determine what is required to be able to write such an adaptor. The first thing we need is the ability to intercept all communication to and from a component, in fact isolating it from its environment. We have to intercept *all* of a component's communication to the outside world, otherwise it may be impossible to adapt the behavior of components. Because open distributed systems are very closely linked to technology, writing components and adaptors are also closely linked to it. Therefore we need to choose a good technology that allows (or enables) us to write adaptors easily.

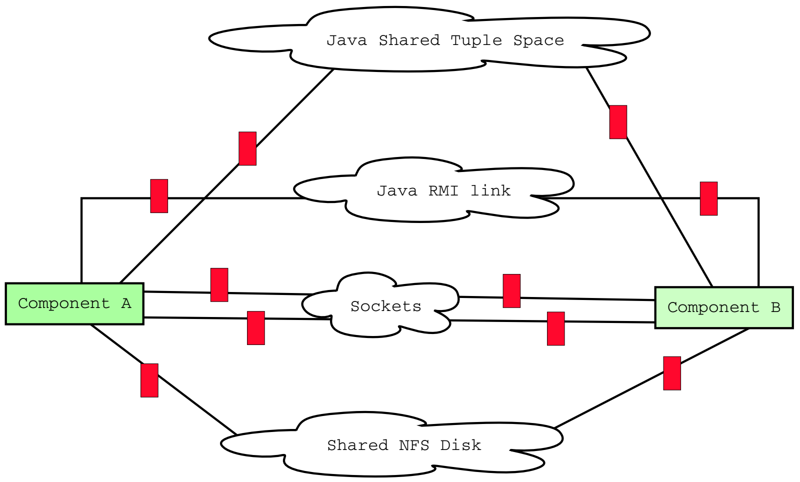


Figure 2.1: This picture shows how difficult it can be to isolate a component from its environment when all kinds of different communication technologies are used. The red blocks show where we need to intercept a connection.

Depending on the technology used, one can have components that can share state, that can communicate with each other by sockets, RMI calls, tuple spaces or shared disks (see figure 2.1). In contrast, we can have a model that simply communicates over a single link (see figure 2.2). It is clear that writing an adaptor for the first kind of technology requires a serious amount of code to intercept all behavior and modify it as necessary, while the second example only requires us to intercept one or two sockets. How we can intercept such a connection and how we can understand what is sent over such a connection will be investigated in this chapter. In general

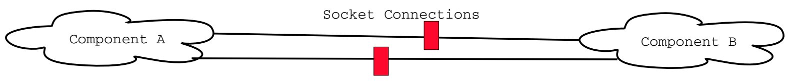


Figure 2.2: This picture shows how one can isolate a component from its environment when it is loosely coupled with its environment. The only links with other components are by means of a (socket) connection.

we will use the SEESCOA component model [SEE99], which is implemented as an event based system that allows us to place adaptors on connections easily. This chapter discusses this model and relates this model to open distributed systems. We will talk about the history of the model, introduce the basic concepts, how services are found, the setup of connections, how communication takes place, management of sessions, concurrency behavior and finally we will explain how one can write adaptors with this model.

2.1 History

THIS CHAPTER USES IDEAS from two event based systems. The first is the mobile multi agent system Borg, the second is the SEESCOA component model.

2.1.1 Borg

Borg [BFDV01] was developed from 1997 up to 2002 by the author of this dissertation. The original goal of Borg was to provide a platform that can run Borg components on all computers that run the Borg virtual machine, providing strong migration and location transparent routing. The system itself is an extension of the Pico [D'H95] virtual machine. Pico is accessible via an extremely simple language, yet its expressiveness is very high, comparable to e.g. Scheme [SJ75]. Pico semantics are defined by a set of 9 evaluation functions that are supported by a storage model and a computational model. The storage model features full storage management and reclamation; the computation model is based on a pushdown automaton that manages expressions and continuations on a double stack. The Borg virtual machine is written entirely both in C and in Borg itself. The user interfaces which accompany the virtual machine run on all kinds of platforms. For Linux users: KDE 1.1.2 and a GNU readline based command line interface. For Macintosh users there is a legacy MacOS 9 version. For windows there is a Windows user interface, which is based on the Cygwin libraries. And finally there is a version running for the Palm Pilot (PalmOS 3.5).

2.1.2 SEESCOA

The second event based system is the component system made for the SEESCOA [SEE99] project. SEESCOA is a project funded by the IWT and 6 industrial partners. The project itself is a cooperation between the University of Gent (UGent), Katholieke Universiteit Leuven (KUL), Limburgs Universitair Centrum (LUC) and the Vrije Universiteit Brussel (VUB). The industrial partners are Phillips, Agfa Gevaert, Alcatel, Barco, Imec and Siemens. SEESCOA stands for Software Engineering for Embedded Systems using a Component Oriented Approach. The component model used in the SEESCOA project is developed by the same team which developed the Borg virtual machine. This component model was written entirely from scratch, using the experience gained from Borg. This allowed us to introduce some necessary semantics that were difficult to capture in Borg due to some implementation issues. The SEESCOA component model is written entirely in Java and focuses on re-usability by means of pluggable adaptors. The fact that the project aims at embedded systems doesn't weaken the component model. Currently, embedded systems need a higher degree of connectivity and, as such, the system itself becomes a distributed system as well. As said above, the remainder of this chapter focuses on the SEESCOA component model.

2.2 The Model

THE SEESCOA COMPONENT MODEL is largely based on the Borg mobile multi agent system. The model itself contains four important concepts, that we explain below:

- ⤳ *Components*: The most obvious concept is a component. A component is an entity that has its own code and data space. Every component has only one execution stack, i.e. components are internally single threaded, which simplifies the model. This implies that every component will handle only one message at a time. Components never share state, they can be compared with processes in an operating system, but without sharing pages.
- ⤳ *Ports*: All components can have ports. Ports are used to communicate between components. A port can accept incoming messages and can send outgoing messages. When a message cannot be immediately sent out the message is queued until the port is (re)connected. If an incoming message cannot be handled immediately, the message is queued as well. A port is the software realization of an interface.
- ⤳ *Connections*: To connect two components with each other, we need to connect a port from the first component to a port from the second component. All connections are full duplex and are typically set up by the component system. A message send is non-blocking and uses pass-by-value.
- ⤳ *Messages:* Components can never modify each other's state directly. The only way components communicate is by means of sending and receiving messages. Messages are self contained, which means that, once they are sent, the content is no longer available to the sender.
- ⤳ *The Component System*: this is the operating environment that schedules the execution of components. It delivers messages between ports and provides a naming service that can be used to identify uniquely components. The component system is responsible for setting up connections. Picture 2.3 illustrates the model.
- ⤳ Within this system an *application* typically is a collection of interacting components, that may be written by different vendors. This implies that a running application is a mixture of controllable and uncontrollable components.

The above model takes ideas from Actor [AMST97] systems, the PI [Mil99] calculus and ROOM [SGP94]. The most important difference with Actor systems is that actors are connectionless, while we do have the concept of connections, which will become very useful, as explained below.

2.3 Naming & Finding Services

AN IMPORTANT PROPERTY of open distributed systems is their highly dynamic nature. In comparison to standard object oriented technology, where a linker glues together all objects of an application before it is started, the programmer has to set up explicitly the links themselves. This is usually performed by specifying information on what kinds of components the programmer is looking for, finding compliant components, and connecting to these components. This addresses the following two subproblems: first, how can one component reference another component, so that they can make contact and communicate and second, how can a component find out which other components offer a certain service.

Before we can contact a component, we need the ability to reference it, just as identifiers in an object oriented language are used to refer and contact objects at runtime. However now we have to take into account that we have multiple applications sharing one global data space. This means that we should be able to refer to components by using globally unique identifiers. Such

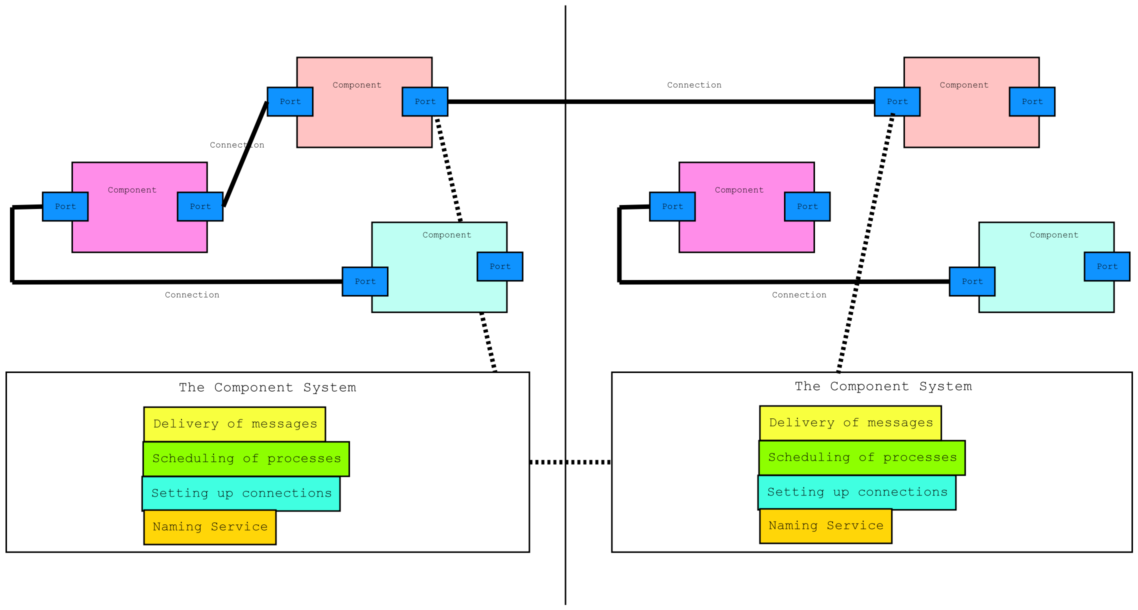


Figure 2.3: The component model as used in this dissertation: the left side of the picture is one host, the right side is another host. The bottom half of the picture shows that we run a full component system on every participating host. The upper half are the components, as executed by the component system. Every component can have ports, ports can be connected with each other.

an identifier should have the same meaning for every component in the system. In other words, we need an identifier that we can use to send messages to the correct place in the network. The most basic idea would be to use the IP-number and the port number of the machine hosting our component and a local component reference. However, this requires the programmer to write down and hard-code some frequently changing *external* information: the unique identifier of another component. It is clear that, to ease development, the system should abstract away from such frequently changing information. Therefore, we use local *ports*, that can be filled in at runtime by the system. We call addressing components through these local ports *implicit* addressing.

Setting up the link between different ports is done by the underlying system, however, in open distributed systems in general, the problem of finding the correct component that offers the required service still remains. To solve this problem directory services (such as JINI [Edw99]) are being implemented. They offer a central point where service-providers can announce themselves, and where service-requesters can look up other providers. This shifts the problem from supplying the correct service to supplying the correct description and looking up services by their description, which is less work because this description can be manipulated at a central place.

As this is not a relevant issue for this thesis, we will not address this subject further, and we will assume that the components already know with whom they will communicate.

2.4 Connecting & Deployment

AS SAID BEFORE, the underlying component system connects different ports to each other using an application specific *connection-broker*. This broker incorporates simple lookup and name to address translation services as well as finding other services by specifying their properties. When deploying an application, this broker will receive an input file that describes the connections between all the components and the links to the external environment.

Connections always take place between two endpoints: we do not support connections between more than two ports. When setting up a connection, the component system will ask both parties to offer a port, based on a description of the required properties of the component. The component then normally sends a *portid* back to the system, which in its turn will use both portid's to connect the ports. Figure 2.4 illustrates how a broker component can request the component system to set up a connection.

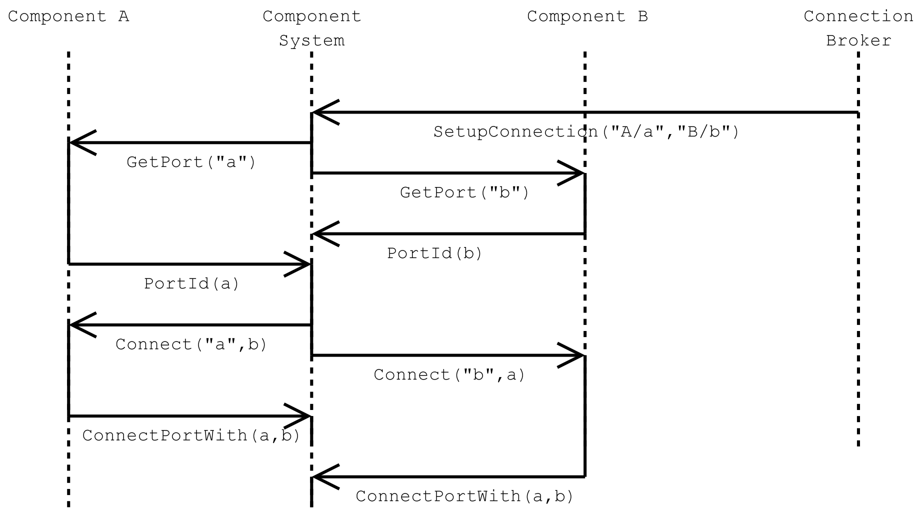


Figure 2.4: Message flow when connecting two components.

From the software development point of view this way of working is very nice. The application programmer points to components and the system wires them together. This offers us the possibility to place adaptors on the connections between components.

The problem with this (and other point and click methods) is their very static nature. Sometimes, we need the ability to receive messages from all kinds of really unexpected components. For example, consider a web server, at component composition time we cannot foresee how many clients, i.e. other components, will join.

Since the base system only allows one connection per port we can only allow a fixed number of clients to join. To address this we have added the possibility to use *multi-ports*. A multi-port is a representation of a collection of ports. One *multi-port* can be connected to a number of other *normal ports*. If we send something to the multi-port, this message will be sent to all the ports in the collection, implementing a multicast.

2.5 Communication

IN THIS SECTION WE EXPLAIN how communication between components can take place. We will explain how messages can be sent and received, the *explicit* representation of messages and why this allows for easy adaptor creation.

2.5.1 Sending and Receiving Messages

At component creation time a component makes a number of ports available for communication. At a certain point the system will connect these ports to ports of other components. Sending a message to another component is performed by offering a message to a local port (which is connected to the port of another component). In the model we use, this is done by invoking the `sendMessage` method upon a port, which will immediately return. At the moment a message arrives on a port `handleMessage` will be invoked on the component. The standard `handleMessage` behavior is to immediately invoke the method corresponding to the message. E.g.: when a message `foo` comes in, first `handleMessage` will be called. If that method doesn't handle the message, the method `foo` will be called. The *only* way in which components can communicate is using through *disciplined* communication: which is sending and receiving messages to each other using `sendMessage` and `handleMessage`

2.5.2 Message Representation

Every message in the system is *explicitly* represented as an association-list of parameters and arguments. In the remainder of the text, we will call the key/value pairs in the association list fields. Fields can be written or read by using the `putField` and `getField` methods of the *Message* class. One predefined field is always present in every message: *Invoke.* This field names the message and is used, in the default implementation, to invoke the correct method upon the receiver. This representation of messages, along with the `putField` and `getField` methods allow an adaptor to handle the messages without needing to know their full content. Messages are deep copied entirely upon sending: a copy of the parameter-strings and a deep copy of all the arguments is made.[1]

The example below illustrates how one can create a message and insert fields. The example also illustrates that the standard `handleMessage` behavior is to *invoke* the *Invoke* field. Hence, `ShowIt()` will receive the message and can retrieve the *Text* field.

```
class MyComponent extends Component
  {
  public Port a;
  public MyComponent()
    {
    a = createPort("a");
    Message msg=new Message();
    msg.putField("Invoke","ShowIt");
    msg.putField("Text","some text to show");
    a.sendMessage(msg);
    }
  public void ShowIt(Message msg)
    {
    System.out.println(msg.getField("Text"));
    }
  }
```

2.5.3 Syntactical Annotation

For clarity, during the rest of this dissertation we will resort to a more simple syntax for communication[2]. Specifying a component is done with the `component` keyword, while declaring a port is done with the `port` keyword. To make sure that an incoming message is immediately invoked upon the component `invoke on this` can be placed behind the port declaration. If a message needs to be handled explicitly by the component, `handle on this` should be used.

To designate a message handler, we use the `message` keyword and to create a field we use a < and > syntax. If we want to read the value of a field we name the field between the `< >`, if we want to set a field we use a `:` (colon). Before the colon we name the field to be set, after the colon we place the value to be assigned to the field. To send a message we use the `..` syntax. The first word following `..` is alway automatically bound to the *Invoke* field. E.g.:

```
component MyComponent
  {
```

[1]Since the "Message" class is the basis for all messages and its standard behavior is to offer an association list it is perfectly possible to optimize local communication by implementing a copy-on-write within the message. For local communication the performance boost is 3 times faster if we do so !

[2]We use a precompiler to translate this extended Java notation to standard Java source code.

```
port a;
message Init()
  {
  a..ShowIt(<Text:"text to show">);
  }
message ShowIt()
  {
  System.out.println(<Text>);
  }
}
```

When programming with such explicit messages, often a type cast is needed to make fields within the message accessible. To help with this, the notation `<type|field>` can be used. This simply expands to `(type)<field>`.

2.5.4 Motivation

This explicit way of sending, receiving and handling messages gives us a greater flexibility when writing adaptors. It allows us to receive all possible messages and handle these without knowing the full message internals. E.g. in case that we want to write an adaptor we can simply override `handleMessage`, ignore the content of the message, but still pass it through. For example: a component placed between two other components which simply prints the messages out and passes them along can be written as follows:

```
component Logger
  {
  port left, right;
  public void handleMessage(Port cameover, Message msg)
    {
    System.out.println("Message "+msg+" from "+cameover);
    if (cameover == left) right..msg;
    else if (cameover == right) left..msg;
    }
  }
```

This simple logger component can be placed between *any* possible two components, without needing to rewrite the Logger component to support new interfaces as they come along.

A second observation about this kind of messages is that this system is truly peer to peer. Any component can send messages to other components, while every other component can receive messages. There is no distinction between *server* components and *client* components. They are all both server and client at the same time. Also, it is not required at compile time to specify with which partners we will connect, this is purely done at runtime.

2.6 Sessions

IN THIS SECTION WE INVESTIGATE ONE OF THE CONSEQUENCES OF NON-BLOCKING MESSAGE SENDS: in an extended conversation between two components, we need a way to explicitly keep track what point in the conversation we have reached, we need to remember session information. However, due the non blocking nature of the communications such session information must be explicitly managed. To allow this, we introduce a new mechanism which easily associates messages with sessions.

2.6.1 Non-Blocking

As said before, the message send is non-blocking, which is a model clearly different from standard object calling conventions. However, this non-blocking model supports open distributed systems very well. Open distributed systems can have long latency times and variable network speeds. Sending a message can be instantaneous are can take an extremely long amount of time, therefore a component working in a blocking way, wastes precious time by waiting for an answer to return. Moreover, since the network is unreliable, we have no guarantee that a return will ever arrive, and therefore we might wait indefinitely.

A non blocking model has none of these drawbacks, however programming in a non-blocking way is not easy. One can no longer simply ask another component something, wait for the reply and continue afterward. To do this one needs to remember what requests have been posted to other components and continue within the correct session when an answer to one of the previous requests arrives. To illustrate the difficulties of such a non-blocking send, consider for example a program that calls 3 components in sequence, where the result of one component is passed to another component. Assuming that a blocking send is available, this could be written in a synchronous way as follows:

```
component Foo
  {
  port a1;
  port a2;
  port a3;
  message Init()
    {
    System.out.println(a3.call(a2.call(a1.call())));
    }
  }
```

On the other hand, if one wants to write this with a non-blocking sending primitive, one should write

```
component Foo
  {
  port a1,a2,a3;
  message Init()
    {
    a1..Call();
    }
  message Result()
    {
    if (port == a1)
      a2.call(<Value>);
    else if (port == a2)
      a3.call(<Value>);
    else if (port == a3)
      System.out.println(<Value>);
    }
  }
```

In this program the `port` field in the `Result` message handler designates the port over which the `Result` message arrived. Clearly, the second program is far more unreadable as the first one. In larger programs the problem of managing different sessions will become even more difficult.

2.6.2 Session Tracking

To address this, a component should be able to map a message to a certain session, and to remember the state of certain values within that session. To do so, we will pass hidden fields along with every message. These fields are passed along automatically when a message is handled and when a new message is sent out. These hidden fields can be used by any component to mark a message, and identify messages when they return. During the rest of the dissertation we will use the > and < notation (instead of '<' and '>') as a syntax for hidden fields. Using these hidden fields is still more complicated than working with non-blocking primitives, however it cleanly separates the session tracking from the application logic. In the example below we see how a session counter (the `>Time<` field) is increased representing a notion of time.

```
component Foo
 {
  port a1,a2,a3;
  message Init()
    {
    a1..Call(>Time:0<);
    }
  message Result()
    {
   >Time: >Time< +1<;
    switch(>Time<)
      {
      case 1 : a2..Call(<Value:<Value>>); break;
      case 2 : a3..Call(<Value:<Value>>); break;
      case 3 : System.out.println(<Value>);
    }
  }
```

2.7 Concurrency

THE DESCRIBED EVENT MODEL uses messages to communicate between different components. Because components are single threaded, concurrency problems within components themselves are avoided, which minimizes the possible places in which they can occur. Now, concurrency problems do not arise from the ordering of statements within the components, but only from the order in which messages arrive. The overall application behavior is uniquely defined by the message sequences. It is clear that this makes this model very suitable for experiments with concurrency management.

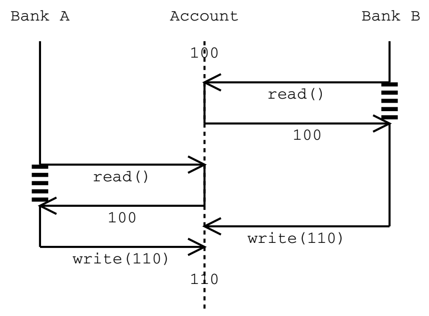


Figure 2.5: Bank accounting example of a concurrency problem

However some message sequences can still give rise to race conditions, deadlocks and other kinds of unwanted behavior. A well known example is the bank accounting example. Suppose we have three components. The first component is a server component which offers two methods: *Read* and *Write*. The second and third components both try to increase the same value at the server component. They do this by reading the value, increasing it and storing the value again. As is shown in figure 2.5, it is clear that the order in which the messages arrives is critical for the correctness of the value.

Classical solution in object based systems such as `synchronized` and thread based mutexes are not applicable in our situation because this is not a thread based model and it is not an object oriented model. Moreover these classical solutions often lead to more problems in the sense that they are difficult to understand, difficult to debug and give rise to a large number of all kinds of inheritance anomalies.

As we will explain in more detail in chapter 5, an important observation is the that the only place where we can solve concurrency problems is within the component itself: the component should offer *locks* for the values that can be updated. In fact, since the component already needs to do some kind of session management when it is accessed from different points, it should at the same time also perform concurrency management. This implies that a component actually offers dual interfaces: an interface for its functionality and an interface for its concurrency strategy.

2.8 Writing Adaptors

WE ALREADY ARGUED that the explicit messaging system offered by the component system offers us a greater flexibility to write adaptors. Above, we illustrated how one can implement a simple logger adaptor that can be placed on any connection between components. We will now further show how adaptors can be written by giving two examples: first we illustrate how we can implement a flow-of-control component, which can be placed at any connection necessary. Second we show how setup and connections of components can be dynamically modified by means of a connection adaptor.

2.8.1 Flow Control by Means of Adaptors

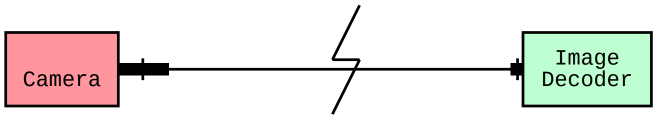


Figure 2.6: A producer that produces data faster than the consumer can consume. This results in overloaded queues at sender side (pictured as the long port at the left side).

The setup in which we will demonstrate our first two adaptor is between a producer component on one machine and a consumer component on another machine. The producer simply grabs images from a camera and sends them out to the receiver. (See figure 2.6). This link goes over a network, so it is possible that the producer produces images faster than the network can handle. This typically results in a producer with overloaded sending queues and eventually an out of memory error on the sending machine.

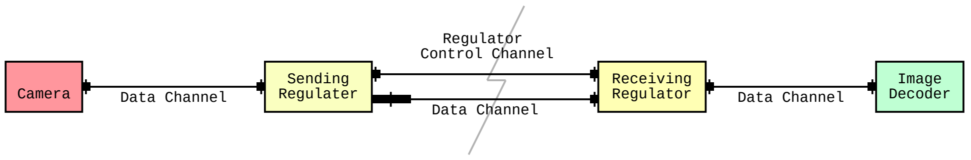


Figure 2.7: A Producer and consumer pair regulated by two adaptors: a sending regulator and receiving regulator. Both regulators communicate over a separate channel.

To solve this problem we need a regulator on the sending side that communicates with a regulator at the receiver side to agree on dropping a certain number of messages. This can only be done of course when the sending side knows how fast the receiving side is receiving messages. This, in turn, turns out to be tricky, because we cannot use the same communication channel we use to send out data: this would place the 'control' messages in the same, overflowing, queue as the images themselves, making regulation substantially more difficult. Therefore we opt for a control channel with a separate queue. An advantage of this is also that the messages on sending side can be simply passed through to the receiving side and vice versa without the necessity to intercept specific control messages.

Algorithm 1 Sending Regulator

```
component SendingRegulator
{
  port camera_side handle on this;
  port decoder_side handle on this;
  port flow_control invoke on this;
  int  imagesSent;
  int  imagesReceived;

  public synchronized void handleMessage(Port p, Message m)
    {
    if (p==camera_side)
      {
      // Image travels from camera side to decoder side.
      if (imagesSent < imagesReceived-10)
        {
        decoder_side.sendMessage(m);
        imagesSent++;
        }
      }
    else
      camera_side.sendMessage(m);
    }
  message FlowReceived()
    {
    // The number of images currently received at receiving side.
    imagesReceived = <Integer|NbReceived>.intValue();
    }
}
```

The sending regulator is given in code in algorithm 1. The sending regulator keeps track of how many message have been sent and how many messages have already been received. If this number is too large new messages are simply dropped. Note that the sending regulator has a method called `handleMessage()` which is used to pass incoming requests from sender to receiver if there is not too much lag. The component also understands `in FlowReceived` messages, which arrive on the `flow_control` port. All messages incoming on the `flow_control` port are automatically invoked, and will therefore not pass through the `handleMessage` routine. This makes writing the adaptor more straightforward as the programmer does not need to differentiate between flow control messages and data messages.

The receiving regulator (algorithm 2) is similar: for every four incoming messages, it sends a flow-control message describing how many data messages have arrived. Messages coming from the sending side are simply passed on to the receiver side and messages coming from the receiver side are simply passed along to the sending side.

To relate this work to existing technologies, such as Java RMI, compare this implementation to how these adaptors would need to be implemented in Java RMI. For RMI, both adaptors should implement the interface of the image receiver. This has two important drawbacks. Firstly, both adaptors are no longer generic because they can only work with the camera-components. Secondly, for each method declared within the interface, a pass-through implementation should be provided, which is tedious work.

Algorithm 2 Receiving Regulator

```
component ReceivingRegulator
{
  port camera_side handle on this;
  port decoder_side handle on this;
  port flow_control invoke on this;
  int  imagesReceived = 0;
  public synchronized void handleMessage(Port p, Message m)
    {
    if (p==camera_side)
      {
      // message is from producer side
      decoder_side.sendMessage(m);
      imagesReceived++;
      // every 4 messages the number of received messages
      // is send back to the sending regulator
      if (imagesReceived % 5 == 0)
        flow_control..FlowReceived(<NbReceived:imagesReceived>);
      }
    else camera_side.sendMessage(m);
    }
}
```

2.8.2 Placing the control flow regulators at runtime

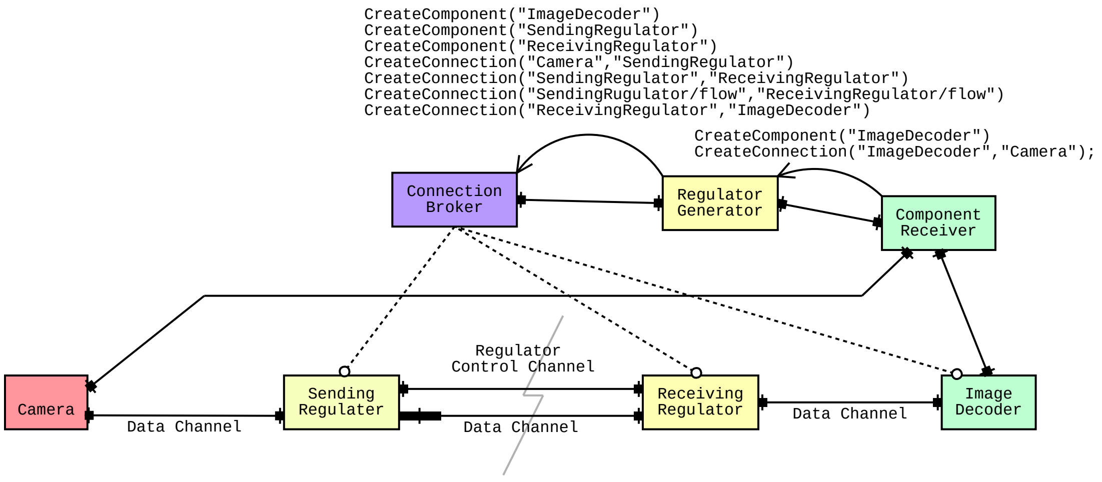

CreateComponent("ImageDecoder")
CreateComponent("SendingRegulator")
CreateComponent("ReceivingRegulator")
CreateConnection("Camera","SendingRegulator")
CreateConnection("SendingRegulator","ReceivingRegulator")
CreateConnection("SendingRugulator/flow","ReceivingRegulator/flow")
CreateConnection("ReceivingRegulator","ImageDecoder")
CreateComponent("ImageDecoder")
CreateConnection("ImageDecoder","Camera");
Connection Broker
Regulator Generator
Component Receiver
Regulator Control Channel
Camera
Data Channel
Sending Regulater
Data Channel
Receiving Regulator
Data Channel
Image Decoder

Figure 2.8: How a regulator generator adaptor can set up adaptors dynamically.

One of the problems often encountered with such setups is that components are created and added at runtime. Assume that the camera is always connected to a *component receiver*, which will create an image decoder when, for example, a new output window of the camera is opened. The component receiver will then automatically set up a connection between the camera and the image decoder. The component receiver uses the connection broker to create new components and set up connections at runtime, therefore it has a connection to the broker. The problem now is placing two regulator adaptors on these dynamically created connections.

Algorithm 3 The regulator generator adaptor

```
component RegulatorGenerator
{
  port controller_side;
  port viewer_side;
  message CreateComponent()
    {
    controller_side.sendMessage(inmessage);
    controller_side..CreateComponent(
      <Instance: <String|Instance>+"Regulator">,
      <BluePrint:"testcases.scss.ReceivingRegulator">);
    }
  message CreateConnection()
    {
    String sending=<String|Id1>;
    String receiving=<String|Id2>;
    sending=sending.substring(0,sending.indexOf("/"));
    receiving=receiving.substring(0,receiving.indexOf("/"));
    sending+="Regulator";
    receiving+="Regulator";
    controller_side..CreateComponent(
      <Instance: sending>,
      <BluePrint: "testcases.scss.SendingRegulator">);
    controller_side..CreateConnection(
      <Id1: <Id1> >,
      <Id2: sending+"/camera_side">);
    controller_side..CreateConnection(
      <Id1: sending+"/decoder_side">,
      <Id2: receiving+"/camera_side">);
    controller_side..CreateConnection(
      <Id1: receiving+"/decoder_side">,
      <Id2: <Id2> >);
    controller_side..CreateConnection(
      <Id1: sending+"/flow_control">,
      <Id2: receiving+"/flow_control">);
    }
}
```

As shown in chapter 3, this can be done straightforwardly by placing an adaptor, which will be called the regulator generator, between the connection broker and the component receiver. We will then not only change the messages sent between the camera and the decoder, but also the messages sent between the component receiver and the connection broker. When a request to generate a decoder arrives at the adaptor (the regulator generator), it will create three new components: the requested decoder, a sending regulator and a receiving regulator. The single connection request from the component receiver, which follows the creation request, is replaced by another setup of connections between

- ⇝ camera and sending regulator.
- ⇝ sending regulator and receiving regulator.
- ⇝ sending regulator, the flow control port and the receiving regulator flow control port.
- ⇝ receiving regulator and the image decoder.

Looking again at Java RMI, there is no standard way to create components at a remote location, and no standard way to make connections between components. Therefore it is simply impossible to write the regulator generator in a generic way, if we would to implement such functionality in Java RMI, we need to define our own standards for component creation and component linking, however this kind of modifications will require the existing components to conform to this standards.

2.9 Summary

IN THIS CHAPTER we have introduced event based systems by means of the SEESCOA component model. We first talked about the history of the model and introduced the basic concepts: *components*, *connections*, *ports* and *messages*. Second, we explained why an implicit addressing scheme is required in a dynamically changing environment, such as open distributed systems. Third, we have shown how the system sets up connections between any two components and how communication between these components takes place. The main ideas here are that messages are represented explicitly and that communication takes place in a disciplined way. Fourth, we explained that the model does not support blocking sends because of the large latency times of open distributed systems, and as a result of this, we asserted that sessions must be managed explicitly. Fifth, we discussed the concurrency behavior of the system and last we showed by means of two real life examples that the system allows for a greater flexibility when writing adaptors.

Because of the flexibility for writing adaptors, the component model presented here is used as the underlying architecture for all our experiments.

Chapter 3

Describing Interfaces by means of Petri-nets

[About Petri-nets] Where is "Start" ??
– Dirk van Deun

WHEN WE WANT TO AUTOMATICALLY GENERATE an adaptor between conflicting interfaces, the program that generates the adaptor needs some knowledge about the interfaces required and provided. This chapter introduces a formal technique to specify interfaces. Specifically we will investigate the use of Petri-nets as a formal documentation technique.

3.1 Introduction

AS EXPLAINED IN PREVIOUS CHAPTER every component has a number of ports. Every port embodies a certain behavior. This can be compared with the interface offered by objects in an object oriented language. This interface, typically called an application program interface (or API for short), must be documented before someone can use the functionality offered by that interface.

Often this API is nothing more than a standard listing of method-signatures. This is clearly insufficient for our purposes for two reasons. Firstly, method signatures have semantics incompatible with the semantics of our ports because we are working in an event based model in which 'messages' are transmitted between components. This implies that messages are passed by value and that a message not necessarily specifies that a certain method needs to be called. A message can be for instance something like $(12, 13)$. Lastly, messages, in comparison to standard method calls, will never return a value. Therefore a simple list of method signatures is not very well suited to document ports. A second reason why common used API's are not suitable is that method signatures do not specify enough information to allow a program to extract interesting properties. This, we will show, is necessary in order to generate an intelligent adaptor. Therefore we will now investigate which technique is usable to formally document an interface.

3.2 Formal Interface Descriptions

COMMONLY USED FORMAL SYNTACTICAL INTERFACE DESCRIPTIONS simply state what methods can be called, with which parameters. Sometimes a type system specifies what kind of objects need to be passed. Specifying interfaces this way suits compiler and linker, and together with an informal explanation of what the interface is supposed to do, can be understood by humans. For machines, on the other hand, there is a lot of information missing within such a simple syntactical description.

- ⇝ They do not specify in *what order* certain methods can be called. This is an important drawback because, without this information an number of possibilities is left open. Possibilities that almost always result in wrong behavior or errors. E.g.: a computer simply doesn't know when a message `Init` should be sent. A human might immediately start by thinking to send an `Init` at the beginning and a `Done` message at the end of some action.
- ⇝ Most of the time formal syntactical interface descriptions only state what they *provide* to a client, they seldomly specify what is *required* from the client. Implicitly the client knows it must be able to handle the return values. In a non-blocking systems on the other hand it is commonplace to use callback messages. These are often specified in an ad hoc way. For example, a whiteboard can decide autonomously to send a `HasDisjoined()` message to the client.
- ⇝ *Session* behavior is almost never specified. Typically a programmer expects an interface to be called only by him. In a non-blocking system an interface may need to process messages in an unknown order. Whether this is possible and how this is managed is also never specified. For example, whether some messages are kept aside and will be processed later is very difficult to express in a formal way.

It is important to observe that the only reason why we are nowadays using formal syntactical interface description is because compilers *need* them. Without them, compilers nor linkers would be able to do either type checking or linkage of two interfaces.

Now, let us think about machines that need to understand interfaces. It is obvious that they cannot make much more sense out of simple API's than they already do (that is type checking and linking). If we want to create an adaptor, then a machine needs to understand enough of the possibilities offered by an interface. Therefore we will do what is typical done is such situations: specify this extra information in a formal way.

The problem that arises now is that, contrary to a syntactical description it is difficult to capture the semantics of an interface. How far should we describe the interaction ? Should we only describe when a certain function can be called or do we also need to specify what the arguments should look like ? If we would specify what the arguments look like do we need to specify the maximum and/or minimum size of the data transferred ? In short, it is very difficult to describe an interface in a formal way without capturing too much detail, or without giving a trivial description (such as: this function will be called at some time). The programmer should have the freedom to specify what he wants in an easy formal way. The formalism should not stop him from expressing certain requirements, it should be flexible and easy to understand. Therefore, the formalism we will use are Petri-nets.

3.3 Petri-Nets

THE FORMALISM WE WILL USE TO SPECIFY THE BEHAVIOR of an interface will be colored Petri-nets. Petri-nets were originally invented by Petri [Pet62]. Petri-nets have a number of very appealing properties. For an in-depth discussion of all these properties see [KCJ98].

1. They are specified by means of a *graphical representation*. A representation that is intuitive and covers in one drawing enough detail to understand what the represented model is about.
2. Petri-nets have a description of both *states and actions*, this in contrast to state diagrams or transition diagrams, which cover only part of the behavior of a system.
3. Colored Petri-nets include *data manipulation* within the Petri-net. A colored Petri-net covers state transition, state of a system and data manipulation in one drawing.

4. Petri-nets are a formalism that can describe a system at *any level of abstraction*. Petri-nets can be used to describe the interaction between high level modules as well as the full interaction within these modules. Petri-nets can specify a large variety of different systems. This can be illustrated by pointing out the number of practical situations in which Petri-nets have helped. see [KCJ98]

5. The *basic building blocks* of Petri-nets are places, transitions (and tokens for colored Petri-nets). These primitives are easy to understand and very *powerful*.

6. Petri-nets allows modularization of systems by means of *hierarchical decomposition*. Petri-nets can be combined using certain operators, which we will not discuss here. For more information see [BFF+95a].

7. For real time systems and timed distributed systems, Petri-nets can be extended with a *time concept*. See [PM93, BMAPY97].

8. *Petri-nets are stable* with respect to minor changes of the modeled system. This is illustrated by many practical experiences. It means that small modifications of the modeled system does not require a complete rewrite of the Petri-net. In many other description languages this is not the case (e.g.: finite automaton).

9. Petri-nets can be *formally analyzed*. This means that certain properties of the modeled system can be verified. This includes: construction of *occurrence graphs* (which global states are reachable), calculation of *invariants* (pre- and post- conditions checking), *reductions* (shrink down a Petri-net but still preserve a number of properties) and checking of *structural properties* (such as starvation).

A large drawback of Petri-nets nowadays is that there is essentially only a graphical notation which is agreed on. A notation in text-format, which is absolutely necessary, is difficult to find and certainly there is no agreement on such a notation. At the end of this chapter we will introduce our own notation, which suits our needs, but before we continue our Petri-net investigation we will look at some other existing techniques.

3.3.1 Related work

It is very difficult to find a formal documentation technique that is a) as general and formal as Petri-nets and b) as useful as Petri-nets at the same time. Below we will shortly touch upon a number of techniques to do so.

State Machines

Finite automatons (FSM's) and state diagrams [Har87, JMW+91, G.01] have problems when multiple concurrent sessions should be expressed and their size explodes very quickly with every newly added behavior.

Reuse Contract

Reuse Contracts [LSMD96], invented at the Programming Technology lab of the VUB, are a means to describe the behavior of an interface in an abstract way. This abstract interface description is called a reuse contract. From reuse contracts a number of properties of an implementation can be deduced, for instance reuse contracts can help in evolving a software system such that existing software dependent on the framework still functions as expected. The approach described in [LSMD96] has some drawbacks, which also form the reason why we didn't use them in our work:

- ⇝ Reuse contracts describe the behavior of non concurrent objects, not the behavior of concurrent components. Whenever faced with concurrent processes the formalism might not be suitable anymore.
- ⇝ Reuse contracts only specify what is provided, barely what is required from another interface.
- ⇝ The level of detail cannot be chosen easily without braking the deductive power.

Message Sequence Charts

Message sequence charts (MSC's) [JCJO92, Wyd01], as a documentation technique, offers example traces of what a component can do. However, message sequence charts typically documents only one run through a component and are difficult to extend to include *all* possible traces of a component. A second drawback of the work presented in [Wyd01] is that it is only possible to reason about the sequence of things to happen, not about the actual content of the data transmitted. For concurrent systems this is a large drawback. It is almost completely impossible to describe the semantics of a rollback-able transaction without taking the *state* of the resources into account.

Pure logical approaches

The use of purely logical approaches that specify what conditions should be met would be possible: it is not too difficult to use predicate-logic or proposition-logic to describe the behavior of an interface to a chosen level of detail. Often this is done by using pre- and post-conditions to describe when a message can be send or received. However, since these approaches are barely readable and do not offer extra advantages over Petri-nets we chose not to use them to describe the behavior of component interfaces.

Temporal logics

The use of temporal logics [KV97, Pnu77] to describe when which transition is enabled could also have been possible. Again the drawback here is the readability and the difficulties one can have to write down even simple statements such as: 'between every occurrence of A and B there can only be one enable or disable transition'.

SDL

Another, well known specification technique such as SDL [OFM97, JDA97], which is widely used to specify communication protocols, can be easily mapped onto Petri-nets [FG98].

IDL's

The lack of a formal semantical description of interfaces in protocols has been recognized for a long time. See [Bra01]. In the past, attempts have been made to extend CORBA IDL's with extra formal specifications in such a way that they could help in automatic checking of the protocols involved [CFP+01]. However, since we are not working with IDL's anyway there was no use in using these approaches. [BOP] discusses how Petri-nets can be used to describe the behavior of CORBA objects.

Existing Petri-net tools

A lot of tools support Petri-net in all kinds of contexts. The main reason why we didn't use them is because we are using Petri-nets in a context in which they are generated automatically. This includes a random element and by generating pseudo-random Petri-nets we would test every tool to its limits. It is not to be expected that a tool which survives all possible Petri-nets will be found quickly. Neither would it be possible to fix bugs in such a tool because most often the source is not free. A second reason why existing Petri-net tools have not been used is that most of the existing tools are to be paid for.

However, there is one tool we did investigate because it looked very promising: *Pep*. *Pep* [Gra] is a programming environment, developed at the university of Oldenburg by the parallel systems group and is based on Petri-nets in which the programmer can design the requirements of a parallel system using a process algebra notation, called $B(PN)^2$ [BH93], SDL [OFM97], high level Petri-nets, called M-nets, or low level petri-nets. Along with the tool comes a set of compilers which generate Petri-nets from the different kinds of input formats. A lot of papers have been published how certain of these languages can be mapped onto Petri-nets [BFF+95b, FG98]. The Petri-nets can be visually simulated, and the possibility exists to link it with a 3D VRML environment. Automatic verification is also included in Pep, by means of an integration of other packages such as an Integrated Net Analyzer, a symbolic verification system (developed at CMU) and others.

Pep's big attraction is due to the good programming documentation and documented internals. The whole abstract Petri-net notation is given in an understandable form. Its drawbacks on the other hand are its incorrect conversion from high level petri-nets to low level petri-nets and its wrong execution of high level petri-nets. It seems as if every high-level place acts as a queue on which messages can come in and whenever a transition needs to be checked only the top level elements are checked. In most cases this is suitable and this is certainly suitable if the Petri-nets are generated automatically. However, a suitable combination of tokens, such as is required from colored petri-nets, is not the case, which makes executing an automatically generated Petri-net an impossible case. Another drawback of Pep is that its Petri-net file format is very rigorous and unreadable. Specifying a Petri-net is difficult because there are cross-linked labels and references almost everywhere. This is something which is a) unnecessary and b) difficult to keep track of when specifying a Petri-net.

A second tool we wanted to investigate, is CPNet. This is the tool promoted by the author of [Jen94], is sold to companies and is supposed to be free for universities. However after contacting the authors 3 times we still didn't get any answer.

3.4 Colored Petri-Nets

PETRI-NETS ARE OFTEN DRAWN AS BOX/CIRCLE DIAGRAMS. Figure 3.1 is a box/circle diagram of a Petri-net describing the behavior of a non-counting semaphore. Petri-nets have a number of concepts:

- ⇝ *Places*: places represent the state of a system. In our example, these are the circles. The places are `Unlocked`, `Unlocking`, `Locking`, `Locked` and `Acting`.
- ⇝ *Tokens*: A place can have zero, one or more tokens. Simple Petri-nets only have boolean tokens. A token is either there or is not there. In our example only the `Unlocked` place contains a token. Tokens which contain values will be discussed when we describe colored Petri-nets.
- ⇝ *Transitions*: A transition specifies how a token is moved from one place to another. In our example, we have the transitions: `UnlockDone`, `Lock`, `LockFalse`, `LockTrue`, `Unlock`, `ActDone` and `Act`. A transition can be either enabled or disabled. If all the arcs coming into a transition offer a token the transition is enabled. In our case only the transition `Lock` is enabled because the only incoming arc from `Unlocked` offers a token. The transition `Unlock` is not enabled since there is no token at `Locked`. When a transition is executed all offered tokens, that take part in enabling the transition, are taken away from their place and transferred to all the places that receive an arc from this transition. If we would execute

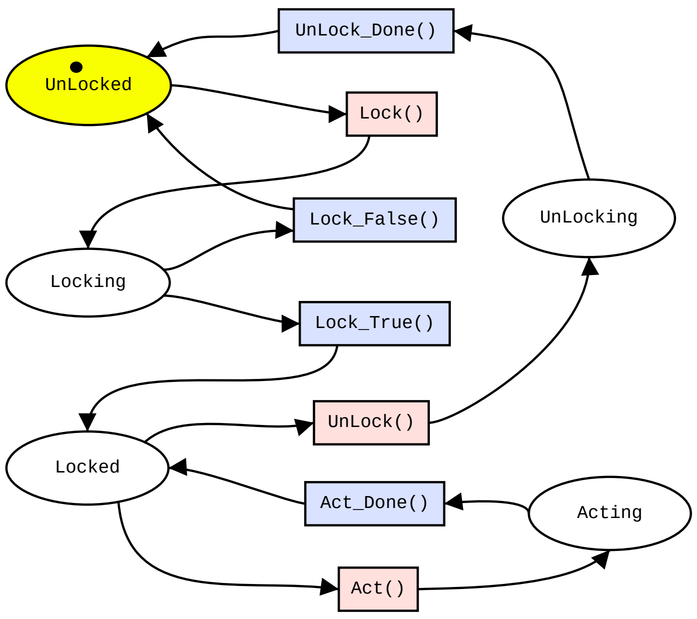


Figure 3.1: A Petri-net describing a non-nested locking strategy.

the transition `Lock` the token would be moved from `Unlocked` to `Locking`. Afterward the transition `LockFalse` and `LockTrue` will be enabled.

⇝ *Marking*: The marking of a Petri-net contains all the places that contain a certain token. The marking is in fact the global state of a Petri-net. It is perfectly possible to have a Petri-net in which multiple places contain a token.

The previous description describes the elementary properties of Petri-nets. Although enough to describe the basic operation of a Petri-net, colored Petri-nets allows tokens to carry a certain value. This small extension to Petri-nets complicates the formalism a lot. It is not clear when a certain transition is enabled: can we specifically check the color of a token or do we only check its presence. It is also not clear what should happen when a transition is executed, what color/value will the outgoing token(s) have ? Is it possible to send different tokens to different places ? Below we will explain how colored Petri-nets are defined.

3.4.1 Informal Discussion

From an informal point of view a colored Petri-net consists of places, transitions, a relation in between them and expressions which are used to verify incoming tokens/values and create new tokens/values. The Petri-net in figure 3.2 illustrates what an easy to understand colored Petri-net looks like for a nested locking strategy in a whiteboard containing 32x32 squares. A number of additional properties can be observed:

⇝ First, every place has a type associated with it, a color set. This type declares which possible values can be present at the given place. For instance the Locking place has a type/color $X \times Y \times C$. X is the set of possible X values $[0..31]$, Y is the set of possible Y values $[0..31]$ and C is the set of possible lockcounts, $[0..\infty[$. The possible colors of the tokens, or values for short, at the locking place are tuples which belong to the set $X \times Y \times C$. The values $(0, 0, 0)$, $(31, 5, 900)$ are valid tokens, while the values $(-1, 0, 0)$, $(32, 5, 900)$ are invalid tokens.

⇝ Second, places can contain more than one token.

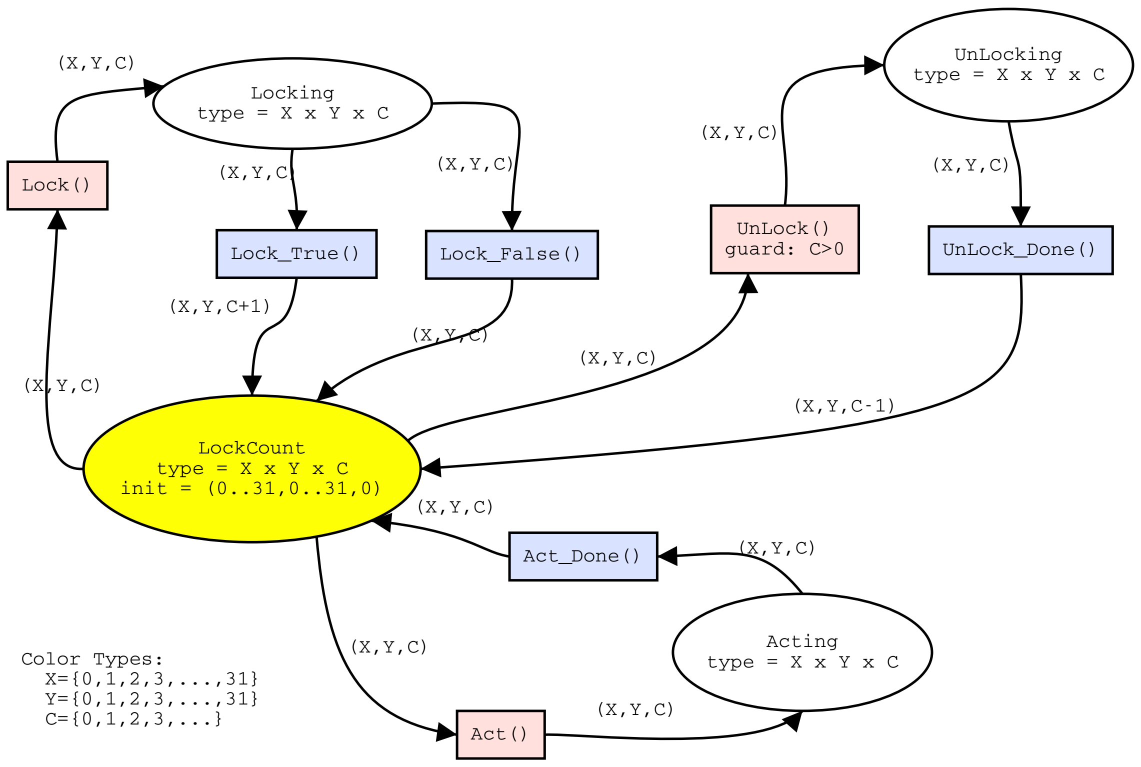


Figure 3.2: A colored Petri-net illustrating a nested locking strategy for a whiteboard of 32x32 squares.

⇝ Third, all arcs contain an expression which either describes the tokens generated or the tokens to be matched. From the point of view of a transition

- every incoming arc describes which tokens are looked for. For example, the incoming arc on the `LockTrue` transition needs a 3-tuple, if one is available, such as $(10, 10, 0)$ the variables X, Y and C will be bound to the values present in the token/tuple. So, $X = 10; Y = 10; C = 0$.
- every outgoing arc describes how new tokens are generated. If the `LockTrue` transition is executed all incoming tokens are removed from the input places and the output places receive newly created tokens. For example, the outgoing arc of the `LockTrue` transition contains the expression $(X, Y, C + 1)$. Since the variables X, Y and C were bound to 10, 10 and 0, the new token will be $(10, 10, 1)$. This token will be put in the place `LockCount`.

⇝ Fourth, some transitions can contain guards. A guard is an expression which verifies whether the transition is enabled given a number of input tokens. A guard is also an expression in some sort of language, which will be described later. A guard should evaluate to true or false. When a guard evaluates to true the transition is enabled, when the guard evaluates to false but still all tokens are present the transition is not enabled. For example, the transition `Unlock` has a guard $C > 0$. Intuitively this means that a lock cannot be released if the lock is not held. The outgoing arc of the `UnlockDone` transition has an expression $(X, Y, C - 1)$ which decreases the lock counter with one. Because we are sure that the incoming token has a lockcount larger than 0, the resulting token will always be in the set $X \times Y \times C$.

⇝ Fifth, the expression language used within guards and on arcs can be chosen. However if one chooses a language too rigorous (Turing complete) a lot of analyzing power might be lost. The language we will choose will be described below.

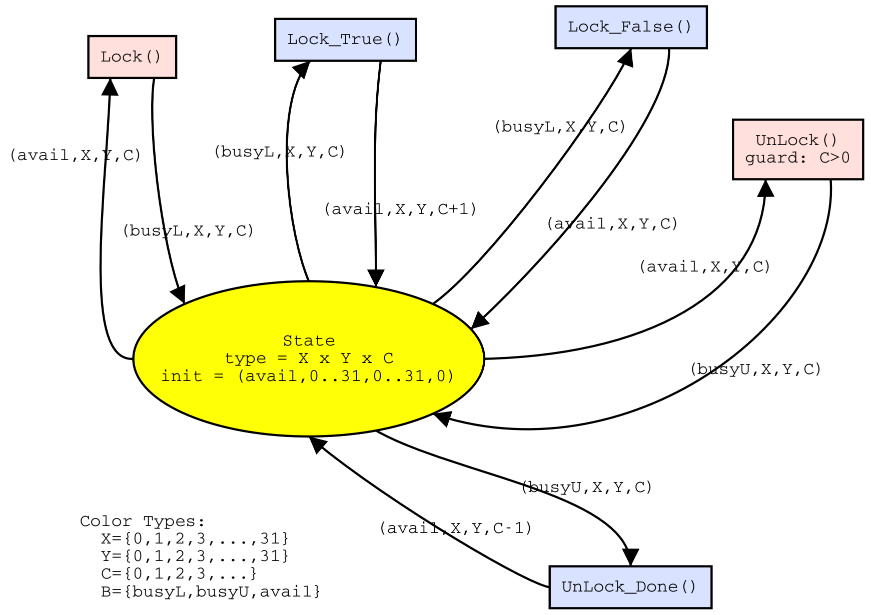


Figure 3.3: A nested locking strategy upon a squared whiteboard with only one place.

Given this, we can now clearly see the expressive power of Petri-nets. We can choose how much detail we include in our Petri-net. The Petri-net given in figure 3.2 only covers the locking of a single square with a lockcount. If we want we could add a session ID to check whether the incoming lock request is from the same one who already has obtained a lock. The fact that we do not *need* to specify this, without losing the ability to reflect over the behavior of the system is one of the greatest strengths of Petri-nets.

It is even possible to describe the same behavior with only one place. Therefore we need to add another color B which describes whether a certain position is in busy locking (busyL), busy unlocking (busyU) or available (avail). The associated Petri-net, without acting logic for the sake of simplicity is pictured in 3.3. On the other hand if we need to use this Petri-net in a larger context, in which we only need to know whether a place is locked or unlocked we can split the `LockCount` state of figure 3.2 in two as depicted in figure 3.4.

Simple Petri-nets, which we will need later on, are Petri-nets in which tokens cannot contain a color. They only can be at a certain place. A simple Petri-net only allows for one token per place. A simple Petri-net also doesn't have guard, input or output expressions.

3.4.2 Formal Definition

We will now define Colored Petri-nets formally. The definition given below is based, to a large extent, on the well known work of [Jen94]. Mainly we have removed some $name \rightarrow place$ and $name \rightarrow transition$ mappings, however this does not modify the formalism it merely simplifies it for our purposes. The often-used and referenced definition given in [Jen94] is not the first definition given of colored Petri-nets, other definitions also exists such as referenced in [EK98, Lak94]. However, they are not explained in as much detail as the one we will use.

Before we can explain the details of colored Petri-nets we need the ability to describe multiple tokens at the same place. Since this is something that cannot easily be expressed with mathematical sets, we resort to multi-sets.

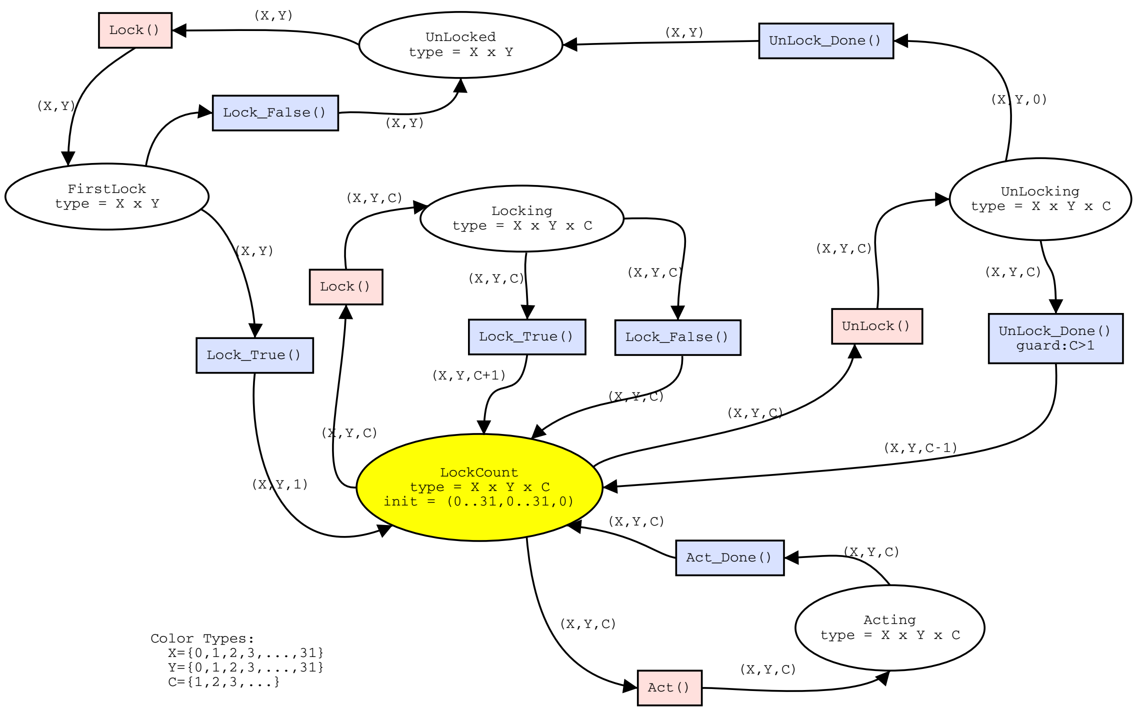


Figure 3.4: A large Petri-net describing the behavior of a nested locking strategy on a whiteboard consisting of 32 x 32 squares. There are separate states for 'unlocked' and 'locked'. When locked a lock counter is kept.

A *multi-set* is a set in which every element can occur multiple times. Formally a multi-set m is defined over a certain underlying set S as a function that maps every element of S to a natural number: $m : S \rightarrow \aleph$. The domain of the multi-set: the occurency counts of every possible element of S are called the coefficients of m. All possible multi-sets associated with a certain set S will be denoted as S_{MS}. This should **not** be confused with the power-set, denoted 2^S which is the set of all possible subsets.

A second preliminary before we can explain the formal side of colored Petri-nets concerns *expressions*. The guards and actions work on values. The way in which these are represented is currently left open, any type of expression can be inserted into a colored Petri-net. For example, one can use λ expressions, or simple algebraic expressions. One can choose whatever fits best. If one chooses a language too expressive a certain level of formal analysis will no longer be possible. We will discuss this later on. Once one has chosen an expression language one cannot change this anymore within the same Petri-net. In the following definitions we will refer to an expression as $expr$. $Expr$ (with a capital) refers to all possible expressions. For every expression $expr \in Expr$ it should be possible to obtain the type and free variables. We should also be able to evaluate it under a certain binding of values to variables:

- ⇝ A type is a finite or infinite set of possible values. For example, a type can be a set such as $\{0, 1, 2, 3, 4, ...\}$ or it can be a set such as $\{'aaa', 'aab', 'aba', 'abb', ...\}$. The boolean type: $\{$true, false$\}$ will be referenced to as 'bool'.
- ⇝ $Vars : Expr \rightarrow Var$ is a function that returns the set of unbound variables within $expr$.
- ⇝ $Type : Expr \rightarrow Type$ returns the set of possible values (the type) an expression can return.
- ⇝ A binding b of a set variables V associates with each element $v \in Var$ an element out of $Type(v)$, such that $b(v) \in Type(v)$.

- ⇝ The value of an expression *expr* under a certain binding b is denoted as $expr\langle b\rangle$. The expression is reduced similar to λ-calculus by substituting every variable $v \in Vars(expr)$ with the value $b(v)$.

We are now ready to define a colored Petri-net formally:
A CPNet is a tuple $N = (\Sigma, P, T, A, C, G, E, I)$ where

- ⇝ $\sum$ is a non empty set of *types*, called color sets.
- ⇝ P, T, A are the *places*, *transitions* and the *flow* relation between places and transitions. $P \cap T = P \cap A = T \cap A = \phi$. The flow relation A contains tuples from $P \times T \cup T \times P$. This is in contrast to the definition given in [Jen94], in which a node-function is added to the Petri-net which maps an arc to such a tuple. This small change however doesn't change any semantics associated with the net as explained in [Jen94].
- ⇝ $C : P \rightarrow \Sigma$ is a color function. This function associates a type with every place. All tokens present at a place p must be of type $C(p)$.
- ⇝ $G : T \rightarrow Expr$ is a guard function if $\forall t \in T : (Type(G(t)) = \text{bool} \wedge Type(Var(G(t))) \subseteq \Sigma)$. Informally speaking, we associate with every transition an expression. This expression should result in a boolean type and all variables used within the expression should require a known type, thus be part of Σ.
- ⇝ $E : A \rightarrow Expr$ is an arc expression, or action: such that $\forall a = (p, t) \in A \vee a = (t, p) \in A : (Type(E(a)) = C(p)_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma)$ where the place p the associated place of a is. The arc expression associates with every arc an expression, which will be used to verify or create new token-values. Every arc expression should evaluate to a set of tokens (a multi-set over the different types allowed by the place). E contains input expressions as well as outgoing actions.
- ⇝ $I : P \rightarrow Expr$ is an initialization function such that $expr \in Expr$ has no free variables and $\forall p \in P : Type(I(p)) = C(p)_{MS}$

Σ	$\{XY = \{0,1,2,...,31\}, C = \{0,1,2,...\}, B = \{busyL, busyU, avail\}\}$
P	$\{State\}$
T	$\{Lock,\ LockTrue,\ LockFalse,\ Unlock,\ UnlockDone\}$
A	$\{(State, Lock), (Lock, State), (State, LockTrue), (LockTrue, State),$ $(State, LockFalse), (LockFalse, State), (State, Unlock), (Unlock, State),$ $(State, UnlockDone), (UnlockDone, State)\}$
C	$\{(State, A \times XY \times XY \times C)\}$
G	$\{(Unlock,' C > 0')\}$
E	$\{((State, Lock),' (avail, X, Y, C)'), ((Lock, State),' (busyL, X, Y, C)'),$ $((State, LockTrue),' (busyL, X, Y, C)'), ((LockTrue, State),' (avail, X, Y, C+1)'),$ $((State, LockFalse,' (busyL, X, Y, C)'), ((LockFalse, State),' (avail, X, Y, C)'),$ $((State, Unlock),' (avail, X, Y, C)'), (Unlock, State),' (busyU, X, Y, C)'),$ $((State, UnlockDone),' (busyU, X, y, C)'),$ $((UnlockDone, State),' (avail, X, Y, C-1)')\}$
I	$\{(State,' (avail, 0...31, 0..31, 0)')\}$

Table 3.1: A Petri-net $(\Sigma\, P, T, A, C, G, E, I)$ of the Petri-net pictured in figure 3.3. The tuple-elements C, G, E and I are functions denoted as a set of couples.

The above describes the static structure of a CPNet. Figure 3.1 describes in a formal way the the Petri-net pictured in figure 3.3.

To describe the dynamic behavior of a CPNet we will first describe what a marking is, then describe when a transition is enabled and finally what happens when a transition is executed. But before we do so, some syntactic sugar is introduced.

$\rightsquigarrow$ $A(t) = \{a \in P \times \{t\} \cup \{t\} \times P\}$ which returns the arcs associated with the transition t.

$\rightsquigarrow$ $Vars(t) = Vars(G(t)) \cup \{v | \exists a \in A(t) : v \in Vars(E(a))\}$

A binding of a transition t is a function b such that

$$G(t)\langle b\rangle \ \wedge \ \forall v \in Vars(t) : b(v) \in Type(v)$$

The set of all bindings for t is called $B(t)$. A token element is a couple (p, c) with $p \in P$ and $c \in C(p)$. A binding elements is a couple (t, b) with $t \in T$ and $b \in B(t)$. The set of all possible token elements is called TE, the set of all possible binding elements is called BE. Now we can define a marking:

A marking is a multi-set over TE. The initial marking M_0 is obtained from I as follows

$$\forall (p, c) \in TE : M_0((p, s)) = (I(p))(c) \tag{3.1}$$

A transition t is enabled if a) there exist a binding satisfying the guard ($b \in B(t)$) and b) all expressions placed on the incoming arcs result in something of the correct type. We will denote this as $M[t\rangle$, which specifies that transition t is enabled under marking M. Formally,

$$M[t\rangle \iff \exists\, b \in B(t) : \forall (p, t) \in A : E((p, t))\langle b\rangle \leq M(p) \tag{3.2}$$

During the rest of the dissertation we will us a shorthand notation $M^\bullet$ to specify the set of all enabled transitions under marking M.

$$M^\bullet = \{t \,|\, M[t\rangle\}$$

$M^\bullet$ will be called the postcondition of M. When an enabled transition t is executed (fired) the marking M_i changes to M_{i+1}as follows :

$$M_i[t\rangle M_{i+1} \iff \forall\, p \in P : M_{i+1}(p) = M_i(p) - \sum_{(t,b) \in BE} E((p, t))\langle b\rangle + E((t, p))\langle b\rangle \tag{3.3}$$

A shorthand notation $M^{\bullet\bullet}$ will be used to denote the set of all possible future markings after firing one of the enabled transitions.

$$M^{\bullet\bullet} = \{N \,|\, \exists\, t \ : \ M[t\rangle N\}$$

For both shorthand notation, $M^\bullet$and $M^{\bullet\bullet}$, which both specify what can happen next, two other notations exist, which describe what could have happened before. ${}^\bullet M$ is called the precondition of M.

$${}^\bullet M = \{t \,|\, \exists\, N \ : \ N[t\rangle M\}$$

$${}^{\bullet\bullet} M = \{N \,|\, \exists\, t \ : \ N[t\rangle M\}$$

3.4.3 Simple Petri-Nets

The above formal definition of Colored *Petri*-nets can be scaled down to Elementary Place Transition nets. To do so, the concept of multiple tokens per place, different colors per token and (input-, output- and guard-)expressions has to be removed. The resulting net has almost the same dynamics as a colored Petri-net, however, because only one token is allowed per place, a transition is only enabled when none of its output places contains a token.

3.4.4 A Note on Implementation

Below we explain that implementing an efficient evaluator for Colored Petri-nets is in general difficult. Later on this might pose some problems when testing verifying certain requirements.

Petri-nets can be implemented on control-flow machines, that is, machines with a 'fetch', 'execute', 'store', architecture. To do so one needs to keep a marking in working memory. With every time-step this marking is used to verify which transitions are enabled. The current marking in the working memory is then replaced by the new marking.[1] However fast in execution, a typical control flow machine suffers from one bottleneck: the memory access: since every single Petri-net step has to fetch and store data in the main working memory, Petri-nets are difficult to map efficiently to commonly used hardware.

Nevertheless, in the past, data-flow machines have been built which are much more efficient in executing Petri-nets. A data-flow machine consists of a number of registers that hold tokens; a token is transferred from operation to operation. In a typical data-flow machine an operation has at most two input-registers and at most two output operations. The input registers are filled in by other instructions that want to pass a token to this operation. The destination registers contain the addresses where to put the result in. These addresses refer to the input registers of other operations. Every operation has also two signaling registers which are used to schedule the passing of tokens. [Moo96] contains a description of a number of existing data-flow machines. Not so strangely the evolution of data-flow machines follows very closely the evolution of formal Petri-net models.

Implementing an evaluator for colored Petri-nets seems trivial: instead of checking whether a token is present and moving the token from the input places to the output places we also have to check a guard. Unfortunately it isn't that simple. Remember the formula 3.2, if we want to know which transitions are enabled we must be able to evaluate the right hand side of that expression. This means that we must find a binding for which the guard (and accompanying expressions) is satisfied. In contrast to an elementary net where we simply have to check whether a token is present we now have to find out which *combination* of tokens is suitable to satisfy the guard. Fortunately, when searching a suitable combination we only need to take into account the set of all tokens present in places local to the transition under investigation. This means that we don't have to check combinations of tokens in places which are not immediately linked to the current transition. This can be easily seen if we look at the two expressions within formula 3.2. The first requirement, (there should be a $b \in B(t)$), does not necessarily guarantee that the values are present in the places local to t. In fact nothing indicates where the values have to come from. It only guarantees that there is a binding which satisfies the guard and which binds all necessary values. The second part of the expression on the other hand $\forall\, (p,t) \in A : E((p,t))\langle b \rangle$ guarantees that all the values necessary to satisfy the guard are present at the incoming places.

In practice this means that, when a transition has x tokens in total over all its input places (this set is called X) and there are y free variables (this set is called Y) we must try out all combinations over X. If there are many tokens this number grows exponentially. Hence we cannot check whether a transition is enabled in $O(1)$ (as can be done with Elementary Nets). A second aspect in evaluating a Petri-net is knowing which transitions have a chance to be enabled. If we start with an initial marking M_0 we know that we only have to check the transitions immediately bound to the places containing tokens. So we only need to check out $M_0^\bullet$. If a certain transition is selected to be executed we need to change the marking from M_0 to M_1 as specified in formula 3.3. In human terms this equation transfers a certain number of tokens from the input place(s) to the output place(s), thereby changing the local states of all the input places and all the output places, formally changing the state of ${}^\bullet t \cup t^\bullet$. The transitions connected to these places: $({}^\bullet t \cup t^\bullet)^\bullet$ are the only ones who can possible change from disabled to enabled (or vice versa).

Because of this, it is important to have Petri-nets with enough distinct places: the number of incoming tokens in a transition will likely be smaller if we have more places in a Petri-net. For example, a Petri-net as in figure 3.3 is very small, with only one place. Given the fact that the net

[1] Instead of testing all transitions to a certain marking it would be more useful to check only all postconditions of a marking.

is conservative[2] and that we start with 3072 tokens, and there are 5 possible enabled transitions, we need to check out 15360 possibilities. This number rises exponentially with the number of input tokens taken by one transition. In comparison, consider the Petri-net in figure 3.2, we start with a token count of 1024 at place `LockCount`. There are 3 possible output places so we need to check 3072 combinations.

Given the fact that we need to try out all combinations of incoming tokens we might start thinking of using something like a logic engine to evaluate a Petri-net. Indeed, it is very easy to write a Petri-net evaluator in prolog [Fla94]. To do so one simply needs to translate the formal definition of a Petri-net to prolog rules as we will do in chapter 8.

3.5 The Expression Language

THE FORMAL DEFINITION of a colored Petri-net given earlier handled expressions in an abstract way. Therefore we need to define what kind of expressions we will use. The syntax:

$expression \rightarrow unaryexpression \mid binaryexpression \mid compoundexpression \mid atomic$
$compoundexpression \rightarrow \text{'('}\ operator\ expression^*\ \text{')'}$
$binaryexpression \rightarrow \text{'('}\ binary\ expression_a\ expression_b\ \text{')'}$
$unaryexpression \rightarrow \text{'('}\ unary\ expression\ \text{')'}$
$atomic \rightarrow \langle integer\rangle \mid \langle variable\rangle \mid token$
$unary \rightarrow \text{'!'}$
$binary \rightarrow \text{' >'} \mid \text{' <'} \mid \text{' >='} \mid \text{' <='} \mid \text{' ='}$
$operator \rightarrow \text{'\&'} \mid \text{'|'} \mid \text{' +'} \mid \text{' -'} \mid \text{' *'} \mid \text{'/'}$
$token \rightarrow \text{'['}\ expression^*\ \text{']'}$

Semantically speaking those expressions are straight forward. In the end everything evaluates to either a token or an integer. There are no other values to work with. If a compound statement is found, the arguments are evaluated in applicative order: they are all evaluated recursively, after that the operator is applied to the given values.

- $+, *$ respectively adds, multiplies all given arguments.
- $-, /$ respectively subtracts, divides all arguments $(-10\ 20)$ results in -10. $(-10\ 20\ 30)$ results in -40. $(/16\ 2\ 2)$ results in 4.
- &, |, ! are logic operations. Something is considered to be true if it is not zero.
- $<, >, \leq, \geq, =$are comparison operators. They are not defined on tokens.

As can be seen, there is no way to store variables, everything is functional. The result of an expression is always the same if the input is the same. No side effects can be specified. The fact that we only work with integers is to reduce the complexity of testing our adaptor. Technically it is not difficult to add other types such as strings, floats, structures and others. The only drawback in doing so is that the implementation of the evaluator becomes much larger and we need to add a substantial amount of type checking.

We now define the $Vars$ operator on expressions.

$$Vars(\ \text{'('}\ operator\ term_0\ \ldots\ term_n\ \text{')'}\ ) := \bigcup_{term=0}^{n} Vars(term_i)$$

$$Vars(\ \text{'['}\ term_0\ \ldots\ term_n\ \text{']'}\ ) := \bigcup_{term=0}^{n} Vars(term_i)$$

[2]A Petri-net is said to be conservative if the total number of tokens present doesn't change over time.

$$Vars(\ \langle integer \rangle\ ) := \phi$$

$$Vars(\ \langle variable \rangle\ ) := \{variable\}$$

The $Type$ operator on expressions is defined as

$$Type(\ '['\ term_0\ \ldots\ term_n\ ']'\ ) := Type(term_0) \times \ldots \times Type(term_n)$$

$$Type(\ '('\ operator\ term_0\ \ldots\ term_n\ ')'\ ) := Type(operator)$$

$$Type(\ boolean\ ) := \{0, 1\}$$

$$Type(\ arithmetic\ ) := Type(\ \langle integer \rangle\ ) := Type(\ \langle variable \rangle\ ) := \{\ldots -1, 0, 1, \ldots\}$$

with *boolean* either $<, >, \leq, \geq =, \&, |, !$ and arithmetic one of $+, -, *, /$. Because all operators work on integers the values assigned to variables can only be integers.

3.6 The Language used to Express Petri-Nets

3.6.1 The Basic Language

WE ARE ABOUT TO USE COLORED PETRI-NETS to describe the behavior of a component. Since a) programmers are supposed to write these and b) our adaptor generation software needs the ability to read and understand them we need a text format to write Petri-nets down in a clear and understandable way, hence:

- ⇝ There should be no redundancy.
- ⇝ Every piece of information that can be inferred should be inferred. This is important because it allows us later on to generate Petri-nets that will have those missing pieces deduced instead of reported as inconsistent.
- ⇝ Things should be written down at the position where people think of them.
- ⇝ The format should be easy extensible.

We looked at a number of existing Petri-net formats, such as the 'Abstract Petri-net Notation' [FKK95], the Pep internal Petri-net format [BG98], and others but none of them suited our needs, either because they were too verbose or because the format was clearly intended for internal use. For instance a number of Petri-net formats describe the incoming and outgoing arcs at different places from the transition connected to them.

Therefore, we came up with the following basic Petri-net syntax. We also implemented a prototype of a Petri-net evaluator based on this syntax in Java. The Petri-nets are based on the expression syntax given earlier.

$petrinet \rightarrow statement$
$statement \rightarrow$ `(place` $variable\ [token]$`)`
$statement \rightarrow$ `(transition` $variable\ inputarcs\ [outputarcs]\ [guard]$`)`
$inputarcs \rightarrow$ `(input` $(variable\,[token])^+$`)`
$outputarcs \rightarrow$ `(output` $(variable\,[token])^+$`)`
$guard \rightarrow$ `(guard` $(expression)^+$`)`

With this notation it is easy to write down Petri-net of figure 3.2.

```
(place LockCount [0..31, 0..31, 0])
(place Locking)
(place UnLocking)
(place Acting)
(transition Lock
  (input LockCount[X,Y,C])
  (output Locking [X,Y,C]))
(transition Lock_True
  (input Locking[X,Y,C])
  (output LockCount[X,Y,C+1]))
(transition Lock_False
  (input Lock_False[X,Y,C])
  (output LockCount[X,Y,C]))
(transition UnLock
  (input LockCount[X,Y,C])
  (output UnLocking[X,Y,C])
  (guard (> C 0)))
(transition UnLock_Done
  (input UnLocking[X,Y,C])
  (output LockCount[X,Y,C-1]))
```

which is quite readable. Normally, colored Petri-nets have places that describe the type of the tokens that they can hold. In our notation we did not introduce types, because often the type of tokens present at places can be inferred from the type of the expression on the incoming and outgoing arcs to/from transitions. In chapter 8 we will indicate how a type inferencer can be written. We will now investigate how we can extend this notation to be more suitable in an adaptor generation context.

3.6.2 Linking Components $\leftrightarrow$ Petri-Nets

Before we can actually use Petri-nets in an execution environment our Petri-nets need to be linked to this environment. Commonly, Petri-nets have a provision for this under the form of sources and sinks. A source is a place where 'out of the blue' tokens can arrive without prior notification. A sink is a place where the Petri-net can place a token and this token will be automatically removed to perform some action.

Informally a source-place is a place for which there are no incoming arcs. A sink place is a place for which there are no outgoing arcs. Formally this can be written down as

$$p \,\text{is}\, source \;\Rightarrow\; \nexists\, t \in T \;:\; (t,p) \in A$$

Similarly,

$$p \,\text{is}\, sink \;\Rightarrow\; \nexists\, t \in T \;:\; (p,t) \in A$$

However, sources and sinks are not only defined based on their presence in the Petri-net. They also have an external behavior associated with them, which is often not expressed within the Petri-net. Below we will define a suitable behavior for sources and sinks such that it can be used within an event based framework.

Integration of Messages

The component framework outlined earlier (chapter 2) describes how ports and components can be used to write adaptors easily. The main idea behind the framework is that messages should be the only way to communicate events between components. Therefore it is a logical extension to link every message to a token. Converting messages to tokens and vice versa is done by a set of conversion rules. Every rule describes what a message looks like and declares how the token should be created. Syntactically we define this,

$message \rightarrow$ `(message` $\langle string \rangle$ $field^*$`)`
$field \rightarrow$ `(field` $\langle string \rangle$ $expression$`)`

Converting a message to a token, given a set of such message rules, is done by retrieving all of the unbound variables, sorting them alphabetically and placing them inside a token. For instance

```
(message LockTrue
  (field "X" X)
  (field "Y" Y)
  (field "Result" 1))
```

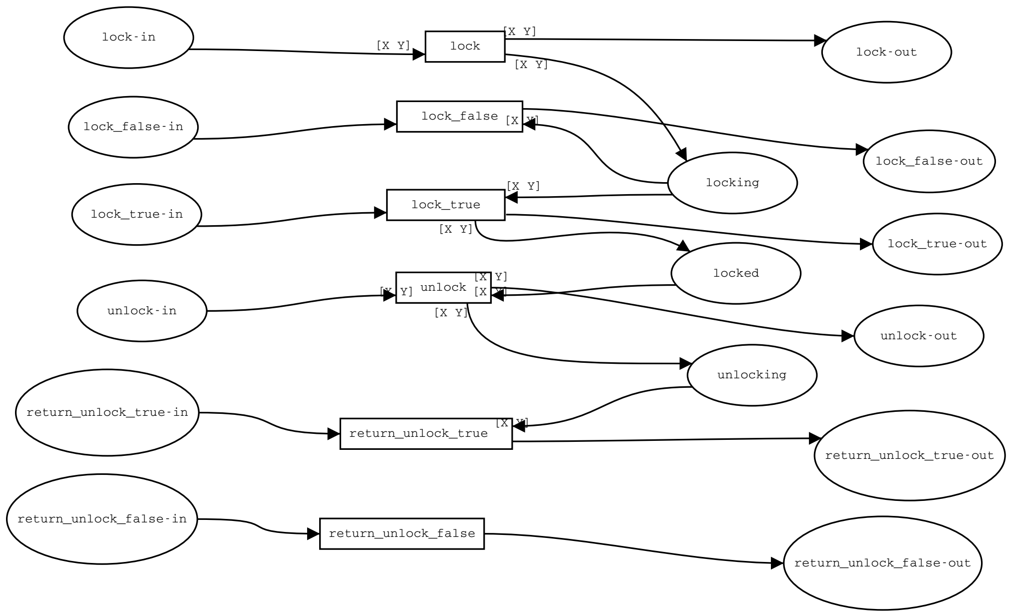


Figure 3.5: An illustration how sources and sinks can be used to interface a Petri-net with a component.

will match any incoming `LockTrue` message and convert it to a token $[X, Y]$. The known `Result` field will not be stored in the token because it is not an unbound variable. For example, `LockTrue(12,15,1)` will be matched by the above rule an will result in a token [12,15]. The `LockTrue(16,18,0)` will not be matched by the above rule, and as such, not generate a token. Likewise, when a token is converted to a message, given a certain message-rule, we simply replace every free variable in the message by the value at the corresponding position within the token.

These message rules give us a means to convert messages to and from tokens. This is necessary to be able to interface a Petri-net with our external component framework. The only thing we still need to define is the way sources and sinks are written down. A source place is a place which generates tokens. In our case, tokens are generated when messages arrive, therefore a source place is described by means of a string (the name), an integer which specified over which port the messages comes from and a message template. Sink places are specified in a similar way.

$statement \rightarrow$ `(source` $variable$ $\langle integer \rangle$ $message$`)`
$statement \rightarrow$ `(sink` $variable$ $\langle integer \rangle$ $message$`)`

The extra $\langle integer \rangle$ field is necessary for sources to specify the port over which the message comes. For sinks it is necessary to specify the port to which the message should go to. Figure

3.5 illustrates how sources (left side, ending on -in) and sinks (right side, ending on -out) can be used to specify a required message interaction between components.

Two Uses for Sources and Sinks

Because this kind of Petri-nets (with sources and sinks) quickly becomes very large, it is impractical to request from the developer to write down all sources and sinks and link them together. However, there are two more reasons why a developer should not write the sources and sinks explicitly down.

1. First, because depending on whether the interface is required or provided the sources and sinks switch places and the Petri-net would need to be rewritten.

2. Secondly, because the sources and sinks can be used to interface a Petri-net between two components, hence link two components together, but they can also be used to link a component to another part of an adaptor.

The second usage will be explained in more detail, after we introduce how in-out transitions are used to describe the linkage with the underlying component.

3.6.3 In/Out Transitions

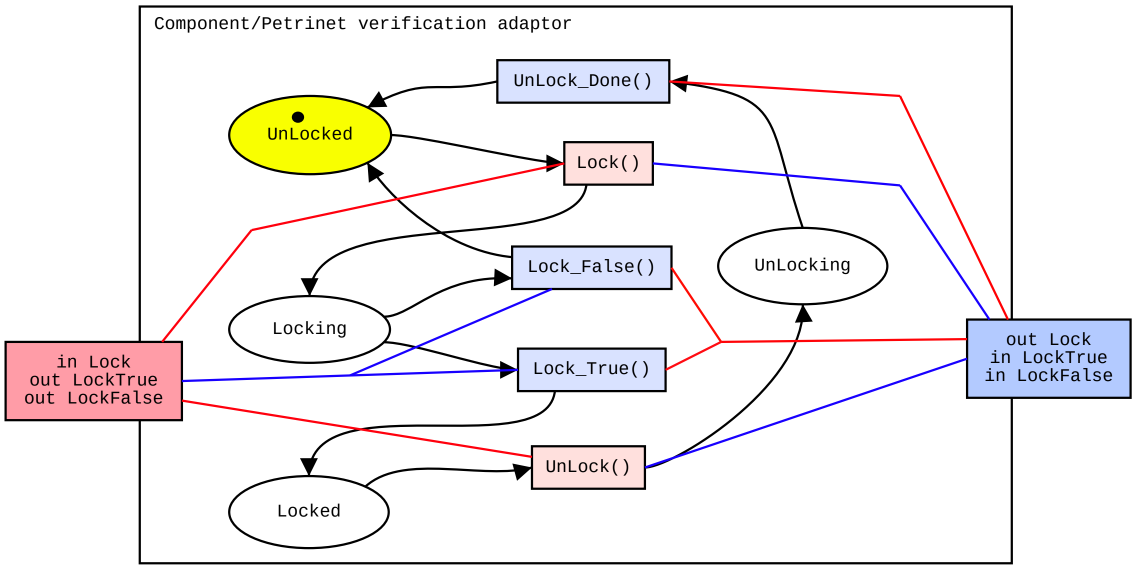


Figure 3.6: A verification adaptor based on 1 Petri-net offered by either server or client.

One of the ideas behind the component framework is that one needs the ability to specify the inverse of an interface. To illustrate this think of a server that offers a locking strategy and a client that expects a locking strategy. The server will specify an incoming `Lock` message and outgoing `LockTrue` and `LockFalse` messages. The client on the other hand will specify exactly the contrary: `Lock` messages go out and `LockTrue` or `LockFalse` messages come in. This can be seen in figure 3.6 and 3.7. The red interface boxes (left) are the ones going toward the server, the blue interface boxes (right) are the ones coming from the client. In practice, the colored transitions are replaced by sources and sinks to match certain messages. The translation from

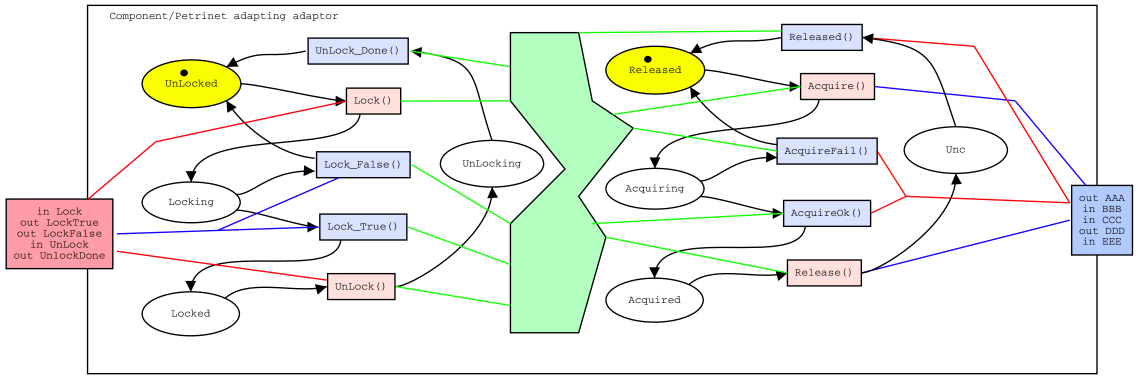


Figure 3.7: An adapting adaptor based on 2 Petri-nets offered by client and server.

such an incoming or outgoing transition to source and sink places is however not always the same because multiple reasons exist to create such an adaptor.

The first situation is one in which an adaptor traces the behavior of an interaction, without adapting anything. This is useful to test the correct working of both components and the Petri-net specification of the interface. Using such an adaptor ensures that the formal description is in tune with the implementation of the component.

A second situation occurs when an adaptor is required to adapt the behavior of two interfaces. In such a situation there will be two Petri-nets available: one accepting messages from the client and one accepting messages from the server. Both Petri-nets and some 'adaption' logic will reside in the adaptor.

These two cases offer some problems because the behavior of the Petri-net transitions is different. To understand this look at the required behavior of the red and blue lines connected to the transitions. Depending on the situation a transition should behave different

1. In the first case, (figure 3.6), a red transition is enabled only if a certain incoming message arrives (e.g. `Lock`) over the *left* port and when triggered a message is immediately sent out to the *right* port.

2. In the second case, (figure 3.7), there are two different behaviors of the red transitions and blue transitions

   (a) the Petri-net tracing the *left* port will only enable *red* transitions when a certain message has arrived on the left port. If such a red transition is executed the adaptor is informed. The *blue* transitions on the left are only executed when the adaptor logic requests so. In response they will send out a certain message to the left port.

   (b) the Petri-net tracing the *right* port will only enable *blue* transitions when a certain message has arrived on the right port. If such a blue transition is executed the adaptor is informed. The *red* transitions on the right are only executed when the adaptor requests so. In response they will send out a certain message to the right port.

As can be seen, the actual Petri-nets generated out of interface Petri-net descriptions might need to do completely different things. Therefore we extended the basic Petri-net syntax with two extra transition types: in-transitions and out-transitions. From the point of view of a component an in-transition describes a message that arrives for the component, an out-transition describes a possible message that can be send out. Since both transitions have to handle incoming or outgoing messages they take some extra arguments.

The extended Petri-net notation:

statement → (`intransition` *variable* *inputarcs* [*outputarcs*] [*guard*] *message*) *statement* → (`outtransition` *variable inputarcs* [*outputarcs*] [*guard*] *message*)

In order to be able to transform the in-transitions and out-transitions to an executable Petri-net we have added special source and sink places (as is done in almost all Petri-net tools). In figures 3.8 and 3.9, the source places are colored green and will contain a token when a message arrives, the sink places are colored red and the underlying component will sent out a message when a token arrives at those places.

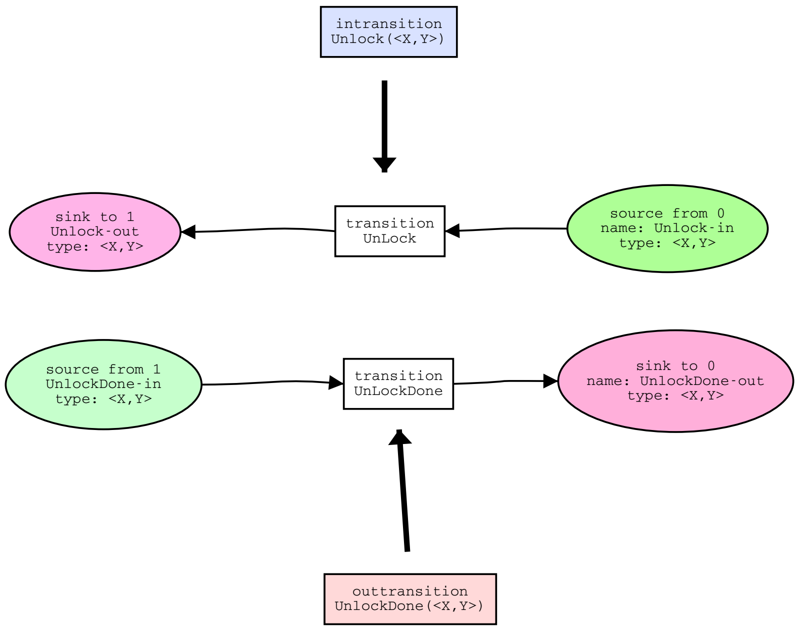


Figure 3.8: Conversion of an in-transition/out-transition to standard sources, sinks and transitions when interpreted as a tracing adaptor.

Transforming a Petri-net to be used in a tracing Petri-net adaptor (illustrated in figure 3.8):

⇝ (intransition *variable inputarcs outputarcs guard message*) is replaced by
(source *variable*-in 0 *message*)
(sink *variable*-out 1 *message*)
(transition *variable inputarcs* ∪ *variable*-in *outputarcs* ∪ *variable*-out *guard*)

⇝ (outtransition *variable inputarcs outputarcs guard message*) is replaced by
(source *variable*-in 1 *message*)
(sink *variable*-out 0 *message*)
(transition *variable inputarcs* ∪ *variable*-in *outputarcs* ∪ *variable*-out *guard*)

Transforming a Petri-net to be used in an adaptor (illustrated in figure 3.9), requires another transformation logic for the in/out transitions; For the left port this becomes

⇝ (intransition *variable inputarcs outputarcs guard message*) is replaced by
(source *variable*-in 0 *message*)
(place *variable*-out)
(transition *variable inputarcs* ∪ *variable*-in *outputarcs* ∪ *variable*-out *guard*)

⇝ (outtransition *variable inputarcs outputarcs guard message*) is replaced by
(place *variable*-in 0)
(sink *variable*-out 0 *message*)
(transition *variable inputarcs* ∪ *variable*-in *outputarcs* ∪ *variable*-out *guard*)

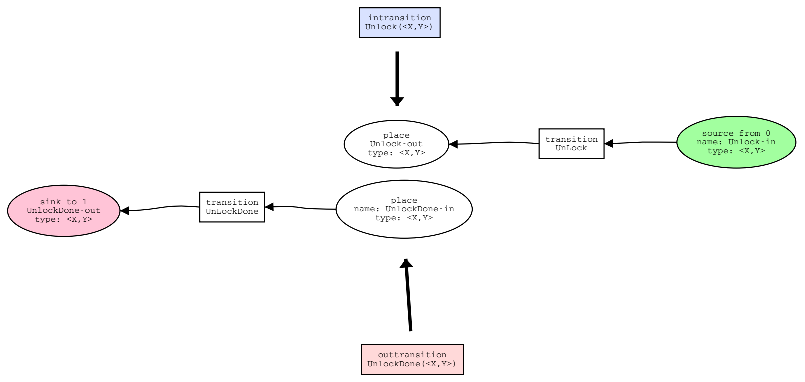


Figure 3.9: Conversion of an in/out transition to sources, sinks, places and transitions when interpreted as an adaptor.

For the right port this becomes:

- ⇝ (intransition $variable$ $inputarcs$ $outputarcs$ $guard$ $message$) is replaced by
  (source $variable$-in 1 $message$)
  (place $variable$-out)
  (transition $variable$ $inputarcs \cup variable$-in $outputarcs \cup variable$-out $guard$)
- ⇝ (outtransition $variable$ $inputarcs$ $outputarcs$ $guard$ $message$) is replaced by
  (place $variable$-in 1)
  (sink $variable$-out 1 $message$)
  (transition $variable$ $inputarcs \cup variable$-in $outputarcs \cup variable$-out $guard$)

By using these in-transitions and out-transitions we have

- ⇝ increased the readability of interface description. Instead of needing to express all input/output places and a transition we now simply write one statement.
- ⇝ increased the flexibility of using these transitions within tools. We can either use the Petri-nets as a tracing adaptor or in combination with another Petri-net as an adaptor.

3.7 Two More Complex Examples

In this section we give two examples of more difficult locking interfaces that can be expressed by means of Petri-nets. The first example covers a layered concurrency strategy. The second covers a rollback-able concurrency strategy.

3.7.1 Interface Description of a Blocking Layered Concurrency Strategy

Figure 3.10 contains an example of a layered concurrency strategy that works in different stages. This example demonstrates how well suited Petri-nets can be to describe subtle interaction patterns between different 'modes' of an interface. The concurrency interface itself describes a) the behavior that can be executed upon certain resources and b) the synchronization behavior that need to be used before these resources are accessible. The concurrency strategy is layered. This means that, before the actual resources can be locked, the server needs to be locked entirely by the client. Once all the locks are obtained the server itself can be released, such that other component might start a set of locking operations. This example has a number of interesting features

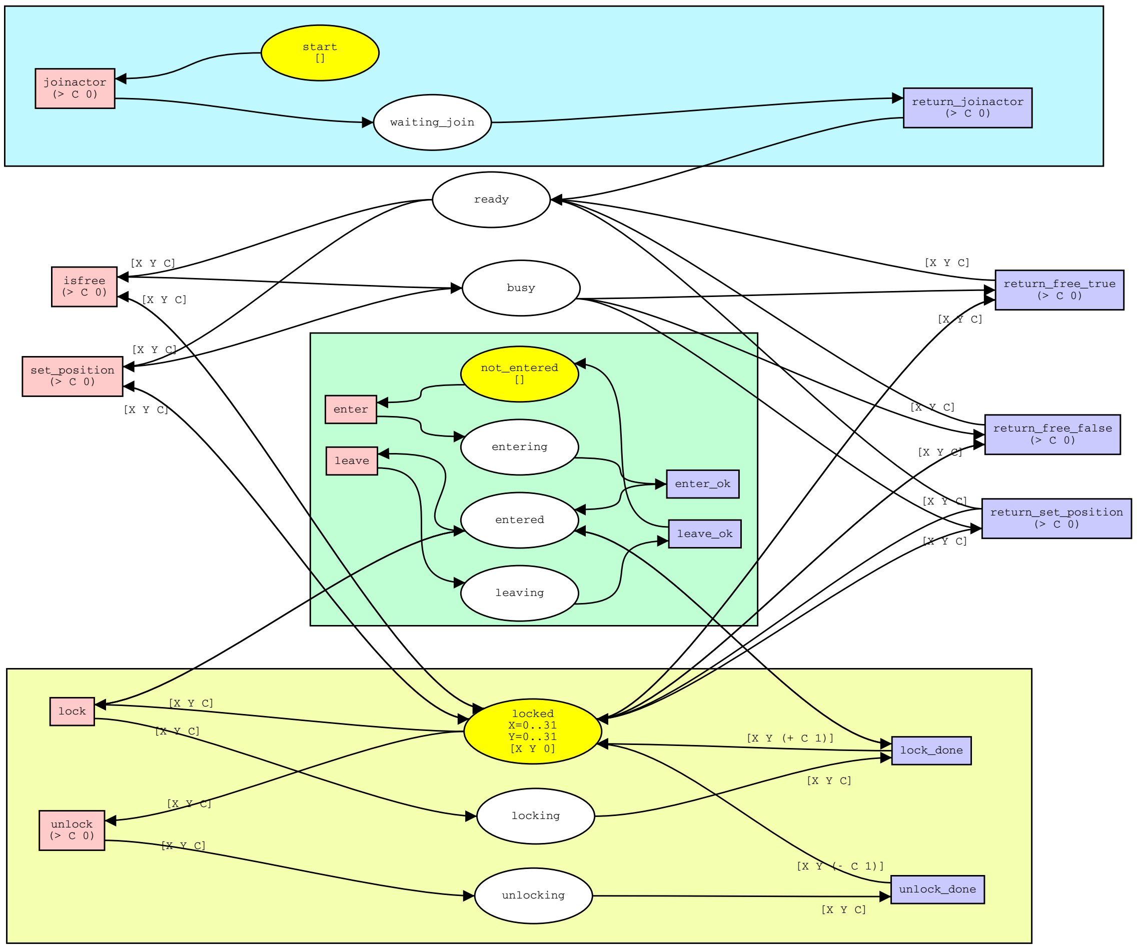


Figure 3.10: A Layered Concurrency Strategy expressed as a Petri-net

Algorithm 4 Petri-net description of a blocking layered concurrency strategy.

```
(place "start" "not_entered" [])
(place "waiting_join" "ready" "busy" "not_entered" "entering" "entered" "leaving")
(place "locked" (foreach X 0 31 (foreach Y 0 31 [X Y 0])))
(place "locking" "unlocking")

     /// SYNCHONISATION BEHAVIOR
(intransition "enter"          (message "StartLocking")
  (input "not_entered")        (output "entering"))
(outtransition "enter_ok"      (message "StartLocking")
  (input "entering")           (output "entered") )
(intransition "leave"          (message "StopLocking")
  (input "entered" [])         (output "leaving") )
(outtransition "leave_ok"      (message "StopLocking")
  (input "leaving")            (output "not_entered"))
(intransition "lock"           (message "Lock" (field "X" X) (field "Y" Y))
  (input "entered")            (input "locked" [X Y C])
  (output "entered")           (output "locking" [X Y C]))
(outtransition "lock_done"     (message "LockDone")
  (input "entered")            (input "locking" [X Y C])
  (output "entered")           (output "locked" [X Y (+ C 1)]))
(intransition "unlock"         (message "UnLock" (field "X" X) (field "Y" Y))
  (input "locked" [X Y C])     (output "unlocking" [X Y C])
  (guard (> C 0)))
(outtransition "unlock_done"   (message "UnlockDone")
  (input "unlocking" [X Y C])  (output "locked" [X Y (- C 1)]))

     /// FUNCTIONAL BEHAVIOR
(intransition "isfree" (message "IsFree" (field "X" X) (field "Y" Y))
  (input "ready")  (input "locked" [X Y C])
  (output "busy")  (output "locked" [X Y C])
  (guard (> C 0)))
(outtransition "return_free_true" (message "RETURN_IsFree" (field "Result" 1))
  (input "busy")    (input "locked" [X Y C])
  (output "ready")  (output "locked" [X Y C])
  (guard (> C 0)))
(outtransition "return_free_false" (message "RETURN_IsFree" (field "Result" 0))
  (input "busy")    (input "locked" [X Y C])
  (output "ready")  (output "locked" [X Y C])
  (guard (> C 0)))
(intransition "joinactor" (message "JoinActor")[2]
  (input "start")   (output "waiting_join")
  (guard (> C 0)))
(outtransition "return_joinactor" (message "RETURN_JoinActor" (field "Result" A))
  (input "waiting_join")  (output "ready")
  (guard (> C 0)))
(intransition "set_position" (message "SetPosition" ...)
  (input "ready")  (input "locked" [X Y C])
  (output "busy")  (output "locked" [X Y C])
  (guard (> C 0)))
(outtransition "return_set_position" (message "RETURN_SetPosition")
  (input "locked" [X Y C]) (input "busy")
  (output "ready")         (output "locked" [X Y C])
  (guard (> C 0)))
```

- It shows how synchronous calls can be written down. This is visualized in the light blue box. The Petri-net starts in the `start` place. Before any whiteboard action (`isfree`, `set_position`) can be executed, the `joinactor` message must be sent. When this has been done, the Petri-net puts a token in the `waiting_join` place. The token that was originally present in the `start` place is no longer there, which means that a `joinactor` message cannot be received anymore. Once the `return_join` message has been sent, the Petri-net will put a token in the `ready` place, which enables all further whiteboard actions in the Petri-net.

- It shows how different modules can be put together. In this example, two synchronization systems are in place. The first offering a server lock (the light green box). The second offering a locking strategy that allows to lock positions on the whiteboard (the light yellow box). The interaction between these two modules is minimal. Only when the server is locked, then can a position be locked (the arrow going from the `entered` place to the `lock` transition).

- It shows how subtle interactions between modules can be expressed. For instance,

  - it is not possible to handle a second lock request as long as the first has not yet been handled entirely (a `lock_done` is send back). In the Petri-net this is expressed by two arcs. The first one pulls a token from the `entered` place and does not put it back. The second one will put it back when the `lock_done` message is sent.
  - Contrary to the locking operation, the `unlock` operation can be executed without a server lock.
  - when a position is locked and the server is *not* locked then it is possible to access the whiteboard with the `set_position` and `is_free` operations, because both operations only depends on the `locked` place and not on the `entered` place.

- It shows on multiple occasions how synchronous calls can be expressed. The places `ready` and `busy` are used by all whiteboard actions (`set_position`, `is_free`, `return_free_true`, `return_free_false` and `return_set_position`). Both places ensure that no new operation will be handled as long as the current operation is not yet entirely processed.

- This example also shows a Petri-net with multiple tokens per place. When initialized, the `start` and `not_entered` place contains a token. Similarly, the `locked` place will be initialized with zero-locks for every available position.

3.7.2 Description of a Non-blocking Rollback-able Concurrency Strategy

The example given in figure 3.11 shows how a concurrency strategy that makes use of rollbacks can be written down. Locks can be requested, when they are granted they can be either committed or aborted. If a lock is aborted the previous state will be recalled, if a lock is committed the previous state is forgotten. Important features of this Petri-net are:

- The `square` place keeps track of the current state of the whiteboard as well as all previous states that it needs to remember. This is done by coupling to every position a color (content of that position) and a version. The type of this place is `[X Y Content Version]`. The current version is the latest version an can be found by looking at the current lock-count. The lock counts are stored in the `lock` place.

- The abort operation works by pulling the current version of a certain position and decreasing the lock counter.

- The commit operation works by pulling a) the current version and b) the previous version of certain place. The content of the current state is put back with the old version number. The lock counter is decreased by one.

Algorithm 5 Petri-net describing non-blocking rollback-able concurrency strategy.

```
(place "square" (foreach X 0 31 (foreach Y 0 31 [X Y 0 0])))
(place "lock"   (foreach X 0 31 (foreach Y 0 31 [X Y 0])))
(place "locking"])

(intransition "lock_request" (message "Lock" (field "X" X) (field "Y" Y))
  (input "lock" [X Y Count])
  (output "locking" [X Y Count]))
(outtransition "lock_false" (message "LockFalse" (field "X" X) (field "Y" Y))
  (input "locking" [X Y Count])
  (output "lock" [X Y Count]))
(outtransition "lock_true" (message "LockTrue" (field "X" X) (field "Y" Y))
  (input "locking" [X Y Count])
  (input "square" [X Y Color Count])
  (output "lock" [X Y (+ Count 1)])
  (output "square" [X Y Color Count])
  (output "square" [X Y Color (+ Count 1)]))
(intransition "commit" (message "Commit" (field "X" X) (field "Y" Y))
  (input "locking" [X Y Count])
  (input "square" [X Y Color Count])
  (input "square" [X Y PreviousColor (- Count 1)])
  (output "square" [X Y Color (- Count 1)])
  (output "lock" [X Y (- Count 1)]))
(intransition "abort" (message "Abort" (field "X" X) (field "Y" Y))
  (input "locking" [X Y Count])
  (input "square" [X Y Color Count])
  (output "lock" [X Y (- Count 1)]))
(intransition "setColor" (message "Set-
Color" (field "X" X) (field "Y" Y) (field "C" Color))
  (input "lock" [X Y Count])
  (input "square" [X Y PreviousColor Count])
  (output "lock" [X Y Count])
  (output "square" [X Y Color Count])
  (guard (> Count 0)))
```

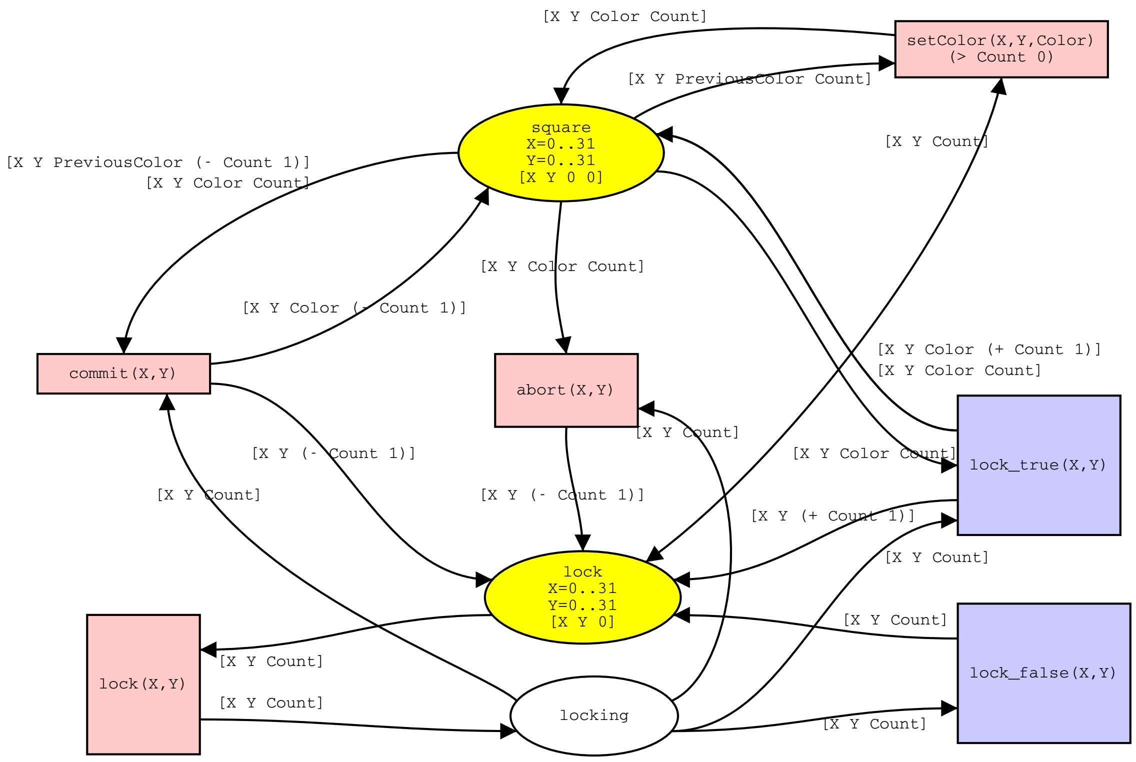


Figure 3.11: A rollback-able concurrency strategy.

⤳ The Petri-net itself is unbounded. This means that, at runtime, the amount of tokens can be infinite. For every acknowledged lock, the `square` place will receive one token extra. Because there is no limit on how many locks can be requested, the Petri-net is unbounded.

⤳ The Petri-net describes an interface that works non blocking. Locks can be queued and handled one by one by the underlying component. If there is no required order upon the message-handling this might lead to disastrous results. For instance, suppose a client has locked 5 times the same position and request two abort and 3 commits. Depending on the order in which the commits and aborts are handled completely different results might occur. Later on we will require that the Petri-net finishes executing before a new message can be handled. This is necessary to avoid this kind of problems.

3.8 The Do-Not 's of Interface Petri-Net Descriptions

PETRI-NETS ALLOW THE PROGRAMMER of a component to document the required and provided interfaces of a component. Nevertheless there are some issues which should be taken into account. Not every Petri-net expresses as much as another Petri-net. For instance, one Petri-net of a component might specify that every incoming message can be handled at any time, while another Petri-net will carefully offer pre-conditions and postconditions for this message to be handled. To help the programmer write down Petri-nets that make sense we introduce some guide rules to write them:

⤳ A Petri-net interface description should always be fully connected. If we take the transitive closure over any transition or place we should end up with the entire Petri-net.[3] This is

[3]The transitive closure does not take the direction of arrows into account.

important because otherwise we would have a set of unrelated message-handlers. A Petri-net such as: if message `a` comes in, then message `a` goes out, if message `b` comes in, then message `b` goes out does not say much more than what a simple syntactical description also says.

- ⇝ There should be no places, other than sink-places, which have no outgoing arcs. Such places introduce some extra information in the Petri-net that can never be used because it is not related to anything else.
- ⇝ In the Petri-net notation introduced in this chapter we never explicitly mention the type of a place, nevertheless the type of tokens that can arrive at a place should be consistent (= the same for every possible transition). Chapter 8.3 contains material that allows one to infer the types within a Petri-net.
- ⇝ All possible incoming an outgoing messages of a component should be documented in the Petri-net description. It is to be avoided at all times to specify only part of an interface. To guarantee this a tracing adaptor is very useful.
- ⇝ A Petri-net description of an interface should be as specific as possible. It should leave as few transitions enabled as possible. For instance, it is always possible to indicate that a message can be handled at any time and return an error when somebody sends this message. This can be described in an interface, but if it is not part of the behavior of the interface it should not be written down. Instead one should write down a Petri-net that clearly disables this transition when the associated action doesn't make sense.
- ⇝ When a message arrives at a Petri-net, at most 1 place should be able to accept the message. With the Petri-net notation described in this chapter it is possible to write a transition which reacts to a message of the type `(message (field "X" 1) (field "Y" Y))`, and another transition that reacts to `(message (field "X" X) (field "Y" 1))`. When a message (1,1) arrives, both transitions will be enabled at once. However, this should not be allowed. With the material in chapter 8.3 it will be possible to check these kind of constraints automatically. However, we will not focus on them anymore.

3.9 Defining Conflicts

By means of Petri-nets we will be able to define clearly what a conflict is. Therefore we will rely on the existence of a certain link between two Petri-nets. This link will consist of the common transitions and will be called the *functional* link. All the other transitions within a Petri-net will be considered to be part of the *synchronization* interface.

We will also assume that both Petri-nets are linked in such a way that the firing of one transition is mapped onto the firing of a similar transition in the other Petri-net (this can be accomplished by inserting an adaptor or by hardwiring the transitions by means of inserting common places, as we've done in section 3.6.3). For instance, when a `SetPosition` message arrives, then the first Petri-net will fire a `SetPosition`. Afterward, the second Petri-net (of the outgoing link) also needs to fire a `SetPosition`, otherwise no communication will occur between both components.

If the Petri-nets are used as such, then we can define a conflict as a situation in which an incoming functional message m can be accepted by the first Petri-net, but not by the second Petri-net because the required preconditions does not hold. E.g, an incoming `setPosition` that cannot be execute on the server because the position is not locked yet (this will be described within the server side Petri-net and will form the blocking pre-condition of the `SetPosition` transition).

Formally, two Petri-nets N_0 and N_1 are, given two markings M_0 and M_1, in conflict when a logic transition exists that is enabled in only one of both interfaces. We will use $\not\approx$ to denote a conflict between two Petri-net markings:

$$(N_0, M_0) \not\approx (N_1, M_1) \Leftrightarrow \exists t \in logic, \mid M_0[t\rangle_{N_0} \wedge M_1[t\rangle_{N_1} \tag{3.4}$$

3.10 A Word on Formal Analysis of Petri-Nets

BELOW WE WILL INTRODUCE a number of formal properties of Petri-nets. Not all of them are decidable.

- **Boundedness**: A Petri-net is bounded if the set of all possible markings generated by a petri-net is *finite*. This property is decidable, it is even possible to check whether the maximum number of tokens arriving at one place doesn't become larger than k. In this case a petri-net is called $k-$bounded. In our situation, we can not always talk about boundedness because the Petri-net descriptions of an interface contains source places that can fire at any moment, so the number of tokens is essentially unknown.

- **Reachability**: A marking M is reachable from marking M_0 under Petri-net N if there exists a sequence of transitions leading from M_0 to M: $M_0[t_0\rangle M_1[t_1\rangle \ldots M$. This is a very important property because it allows us to decide whether a certain error condition can be met or not. Recursively we can define this as follows:

$$M \text{ reachable under } N \text{ from } M_0 \iff$$

$$M = M_0 \vee \exists M' \in {}^{\bullet\bullet}M \mid M' \text{ reachable under } N \text{ from } M_0 \tag{3.5}$$

This property is decidable and forms the basis for many other properties. In chapter 8 we will explain in more detail how this kind of information can be obtained.

- **Deadlock-freedom:** A Petri-net is deadlock free if every reachable marking enables some transition. This property is decidable because it can be reduced to the reachability problem. This definition will later on give rise to our notion of application deadlocks within our event based system. (section 5.2 on page 80)

- **Liveness**: A Petri-net is alive if every transition can always occur again. Whether this is decidable or not is still an open question. This will turn out to be an important property, because it tells us that a certain concurrency strategy does not lock out certain functional behavior. For bounded Petri-nets it is known that liveness is decidable because it reduces to reachability. However, because we cannot always rely on boundedness because our Petri-net can receive a token at any moment, we must assume that this property is undecidable.

- **Homestate-problem**: A Petri-net has a home state M if this marking can be reached from every other marking. The question to decide now is whether a given marking is a homestate for a certain Petri-net. This problem has been shown to be decidable.

- **Non-Termination**: The question whether a Petri-net will never terminate is undecidable.

For a survey on the decidability of Petri-nets see [EN94]. All the above properties are valid on elementary nets and certain colored petri-nets, depending on the expression language chosen within the colored net. However, a lot of petri-net variants exist which somehow introduce a check for the absence of a token. In such Petri-nets a transition can be enabled if there is *no* token at a certain place. If such a construction is added to a Petri-net one loses all decidability as is shown in chapter 7 of [Pet81].

3.11 Summary

IN ORDER FOR AN ADAPTOR to mediate the behavior of two conflicting interfaces, it needs extra formal documentation. The formalism we will use to represent the behavior of an interface are Petri-nets because it allows us to

- ⇝ Express interfaces in a natural way.
- ⇝ Express an interface up to a chosen level of detail, but still complete.
- ⇝ Simulate the behavior of an interface without actually having the implementation of the interface at hand.

After having introduced Petri-nets we explained the link between Petri-nets and the event based model of chapter 2. This is done by means of in- and out- transitions. Depending on the situation we can translate these special transitions to other Petri-nets. We gave two examples of this: a tracing Petri-net adaptor between interfaces which are known to work together and an adaptor between two different interfaces. We also gave an overview of possible formal analysis that can be carried out on Petri-nets.

We wrapped up this chapter with giving some guidelines to the developer to write Petri-nets and gave a short introduction to which properties are decidable for Petri-nets.

The Petri-nets we have described here will be used further on for two purposes:

1. Describe the behavior of the components in an understandable and useful way.
2. Describe the behavior of a liveness module.

Before we do so, we will focus on the last of the preliminaries: learning algorithms.

Chapter 4

Learning Algorithm

THE FINAL CHAPTER OF THE PRELIMINARIES gives an overview of learning algorithms and how they can be used to mediate the semantic differences between different concurrency strategies. Given the context of open distributed systems, this problem cannot be solved automatically. This is because we cannot expect to have an adaptor available for every possible pair of conflicting interfaces, so we *need* some automatic approach.

We have a problem in which we need to create an adaptor that is able to establish meaningful communication between two components with a conflicting interface. As already mentioned in the introduction, the adaptor will be divided in three modules. One module, the liveness module, will be responsible for keeping the underlying component alive, a second module, the concurrency module, will be responsible for implementing a suitable concurrency strategy and a third module, the enforce-action module, is responsible for bypassing an existing concurrency strategy. The second and third modules can be implemented in a formal way, the first module however is computationally too complex to be able to handle formally (see liveness of Petri-nets in section 7.3.1). Therefore we will resort to a learning algorithm.

Before we can look for solutions to the problem presented we need to investigate which techniques we can use to keep the underlying component alive. We need to know what we can expect from a given learning algorithm and what kind of information it expects. In this chapter we first explain a number of important concerns with respect to learning algorithms in general, after which we will briefly introduce a number of different learning algorithms available. Secondly, based on this information we will explore the possibilities we have to map our problem on existing learning algorithms. Thirdly, we explain how genetic algorithms and genetic programs are conceived, and how we will use them in this dissertation. Fourthly, we present reinforcement learning algorithms.

4.1 A World of Variations

A LEARNING ALGORITHM IS CHARACTERIZED by the sort of problems presented, the sort of solutions to be found and the environment in which the algorithm is supposed to work. For instance, in our case we have a problem of conflicting interfaces: the solutions we are looking for are adaptors between the conflicting interfaces and the environment includes all the limitations we place upon the adaptors. Examples of such limitations are 'an adaptor is a process', or 'an adaptor can only handle one message at a time' to 'the conflicts we are dealing with are concurrency problems'. All these implicit or explicit degrees of freedom define the search space of the learning algorithm.

Sadly there is no universal learning algorithm that will work optimally in every possible situation. Therefore learning algorithms are designed to work within certain environments. Learning algorithms cannot learn what they cannot see, and they don't work very well if they see too much information. (This is the typical bias versus variance problem). This is a bit of a disappointment for people introduced to learning algorithms because they assume that an intelligent algorithm should be able to look for other clues than the ones found in their environment. Learning algorithms will not exhibit such behavior [Mor96]. Nevertheless they can be helpful in certain domains such as character recognition, robot motion planning, robot activator control, regulation systems and others.

The number of available learning algorithms is large, and the number of variations on every algorithm itself is even larger. We can consider different criteria for classifying learning algorithms. We will discuss them below:

- ⇝ *On-line/off-line learning*: Some learning algorithms need to be trained *off-line* to learn how to react in certain situations. Once this is done the resulting configuration is fixed and used in an *on-line* situation. Consequently, when new problems occur the algorithm will not adapt itself further. Other algorithms on the other hand will immediately learn what to do in an on-line situation and doesn't require a separate off-line training phase.
- ⇝ *Supervised/non-supervised* learning: supervised algorithms require categorized examples. After presenting these examples to the algorithm, adaptations are made to the configuration such that the different categories are recognized correctly in the future. With non-supervised learning, there is no explicit set of good and bad examples.
- ⇝ *Classifier/parameter tuning* algorithms: Classifier algorithms are algorithms which learn to classify certain input into categories. On the other hand, some learning algorithms are designed to maintain a certain global requirement, which is influenced by a number of different interacting parameters. Parameter tuning algorithms are useful to find out the correct values for these parameters.
- ⇝ *Basic → general* and *general → basic*: Within the category of classifier algorithms two sub categories can be defined: basic to general learners and general to basic learners. The basic to general approach will try to find a suitable descriptive feature of a set of examples such that these examples belong to the same category. A general to basic approach initially places all examples in the same category, and afterward tries to split large categories into more specific, smaller sub-categories.
- ⇝ *Numerical/symbolic learning:* certain learning algorithms work with integer or float values to represent the environment, while others work with symbols (discrete values) to do so. Numerical approaches are very well suited in regulation systems.
- ⇝ *State/stateless*: some learning algorithms are stateless in the sense that they cannot remember previous events. Other algorithms do have state and can, when necessary, recall a previous event.
- ⇝ *The biologically inspired model they simulate:* these can be genetic algorithms, neural networks, reinforcement learning or others.

Because we have a large number of criteria for categorizing learning algorithms, in practice selection of a suitable algorithm is a very complex process. Somewhere, there is an abstract notion of a learning algorithm (a neural network, a genetic algorithm, a classifier system) which is made very specific, depending on the situation in which it should work. Therefore we will investigate the applicability of a number of learning algorithms by tuning certain aspects of the algorithm. However, before doing so, we present a number of different learning strategies:

- ⇝ *Tree Searches:* Breadth-first, depth-first, priority-first searches: by searching the entire solution space we are almost guaranteed to find a the best solution. The environment is presented in an abstract form, the possible actions are enumerated and the possible outcomes are defined. By checking every possible solution the algorithm can finally come up with the best answer. The problem here is that most often, the search space is too large to be able to find anything useful within a reasonable amount of time. Nevertheless, this technique is often used: chess-programs with a minimum-maximum strategy are an example. Usually, programs relying on this kind of problem-solving introduce some kind of pruning: a method of weeding out complete branches that are assumed not to contain useful solutions. However, using pruning is a dangerous and very delicate technique that requires excellent domain knowledge.

⇝ *Genetic Algorithms/Programming:* Genetic algorithms, which are inspired on a model of evolution as defined by Darwin [Dar59] (the survival of the fittest) and Mendel [Gre65] (properties of sexual reproduction). The idea behind genetic algorithms is that a population of solutions is measured in a certain environment. The test assigns a fitness to every individual. An individual that is not fit enough dies and is replaced by a child of 2 individuals with a high fitness. This way (and by means of mutation), new individuals are created and the overall fitness of the population rises. Genetic programs are genetic algorithms in which the individuals are programs and fitness is measured by executing the programs.

⇝ *Reinforcement learning* [SAG98] is an on-line technique. In contrast to the above methods, reinforcement learning algorithms are defined by the kind of problems they solves. A reinforcement learning problem is a situation in which a learner needs to learn which actions should be taken to maximize the received reward in the long run. Typically there is an interaction between the actions, the environment and the rewards: after taking some action, a, possibly delayed, reward is assigned and the environment will probably change. Based on the reward (or punishment) the learner receives, its behavior will be altered in the future. A typical problem of this approach is the tradeoff between exploring the environment and exploiting its knowledge.

4.2 Mapping our Problem onto a Learning Algorithm

TO CHOSE A SUITABLE LEARNING ALGORITHM for our problem, we need to define the boundaries clearly.

⇝ On-line/off-line: Whether the adaptor should be trained on-line or off-line is an important question because of the nature of open distributed systems: the behavior of components may change at any time. We could argue that training an adaptor off-line in a real world setting, with the ability to reset participating actors, and afterward inserting it in the running system would produce a suitable result. However, this is not the case because most often an off-line trained adaptor behaves statically when inserted in the running system: it will not adapt to previously unencountered behavior. This means that when such a situation occurs the adaptor is completely at a loss and won't work properly. Therefore we need an on-line learner. However, when we use an on-line algorithm we should be absolutely sure that the algorithm doesn't place participating components in an unstable state. Hence it should only try out things that are allowed in the given context.

⇝ Supervised/non-supervised: The algorithm should work in a non-supervised way, simply because we cannot define a suitable supervisor in an open distributed system. Furthermore supervised learning is usually performed off-line, which we have chosen not to do. For the sake of the argument, assuming that we could use an off-line learning strategy, then still it would be very difficult to give specific examples of good or bad concurrency strategies. We will come back to this issue in chapter 7.

⇝ Symbolic/numerical: a characteristic of our case is that the environment is, to a large extent, symbolic: the actions presented at an interface are discrete. The difference between a `JoinActor` message and an `IsFree` message is as important as the difference between an `IsFree` message and an `Unlock` message. Therefore we need a learning algorithm which works on *symbolic* input and *symbolic* output. Processing the environmental input can be either symbolic or numerical. However, because the nature of the interface adaptors is typically a problem of interaction conversion: 'when to send what', processing the input is preferably symbolic.

Given these characteristics of our problem, *reinforcement learning* seems to be a correct choice to match our requirements. However, other approaches such as priority-first searches and genetic algorithms will become very useful to investigate the power of certain representations. Therefore, in the sections below we will introduce genetic algorithms and discuss the basics of a reinforcement learning technique.

4.3 Genetic Algorithms

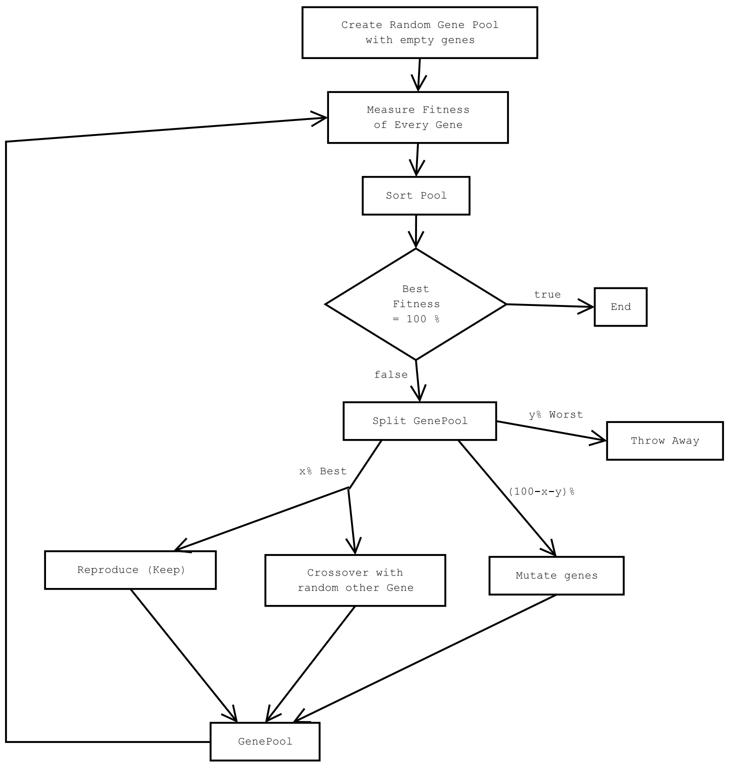


Figure 4.1: A Genetic Algorithm

BELOW WE EXPLAIN THE DETAILS OF A GENETIC ALGORITHM. We will also explain how we use them as representational testers in this dissertation.

A genetic algorithm [Gol89] is an algorithm that tries to solve a problem by trying out a number of possible solutions, called *individuals.* Every individual is an encoding of a number of modifiable parameters, called *genes,* and is assigned a *fitness* that measures how well the individual solves the problem at hand. From the pool of individuals a new *generation* of individuals is created. This can be either by *preserving*, *mutating* or *crossing over* individuals. This process is repeated until a suitable individual is found. An illustrative flowchart of this process is pictured in figure 4.1.

The standard questions before implementing any genetic algorithm are: What are the individuals and their genes? How do we represent the individuals? How do we define and measure the fitness of an individual? How do we initially create individuals? How do we mutate them and how do we create a cross-over of two individuals? How do we compute a new generation from an existing one? At the moment a genetic algorithm is implemented in a way that the individuals are themselves programs, we call it a genetic programming algorithm. In our implementation, the individuals could be protocol adaptors between communicating processes.

4.3.1 Genetic Algorithms as Representational Testers

Genetic algorithms as an evolutionary algorithm to solve the problem of conflicting concurrency strategies is relatively useless in a running system, because a genetic algorithm works in general off-line and can never guarantee that the resulting adaptor will work in every possible situation. Indeed: it is very difficult to verify certain concurrency requirements such as deadlock-freedom or no race conditions. However, we will use genetic algorithms is such a way that we turn an argument against them to our advantage. This argument is the *representational problem*: how well does a genetic algorithm perform given a certain representation of the problem [SB92,Sch87]. Or, stated otherwise how can we create a representation that will offer a good fitness landscape.

To illustrate this, think of a problem in which an ant has to learn how to walk a line from left to right. If we use a representation of our individuals in which we only offer the generation process 'go left' and 'go right' primitives, the genetic algorithm will very easily find a solution to the problem. However if we use another representation of the individuals with operators such as 'current angle', 'current position', 'turn left x degrees', 'turn right x degrees', 'forward' and 'backward' it will be substantially more difficult to find a solution to the same problem. We will use this property of genetic algorithms (which is, in general, a big disadvantage), to determine the best representation of a solution within a given environment and fitness function. Specifically, we will keep the same environment and fitness function and will measure how fast a solution is found given a certain representation of the individuals. We consider that if a certain problem can be solved easily under a certain representation, it is a good representation. If the same genetic algorithm has more trouble finding a solution the representation might not be so suitable. In chapter 9 and chapter 11 we will use this technique to validate different representations.

However, which kind of representation are available for our problem still needs to be discussed. In general two approaches are used. First, we can use hierarchically structured, human readable/programmable programming languages such as Scheme, Lisp, Java and others to represent the behavior of the concurrency adaptor. Unfortunately, the inevitable syntactic structure imposed by these languages complicates the random generation of programs. Second, we can use a representation that is more suitable for random generation, but is in general not readable. These approaches are mainly based on classifier systems, for which we will explain the basics below.

4.3.2 Classifier Systems

We will now introduce one of the representations we have tested because we need them in chapter 11. This representation are classifier systems. Classifier systems [Gol89] in cooperation with genetic algorithms form a genetic programming[1] approach that is symbolic and easy to implement. Moreover, given our problem, their symbolic nature can be to our advantage.

A classifier system is a kind of control system that has an input interface, a finite message list, a classifier list and an output interface. The input and output interfaces put and get messages to and from the classifier list. The classifier list takes a message list as input and produces a new message list as output. Every message in the message list is a fixed-length binary string that is matched against a set of classifier rules. A classifier rule contains a number of (possibly negated) conditions and an action. These conditions and actions form the genes of each individual in our genetic algorithm. Conditions and actions are both ternary strings (of 0, 1 and #). '#' is a pass-through character that, in a condition, means 'either 0 or 1 matches'. If found in an action, we simply replace it with the character from the original message. Table 4.1 shows a very simple

[1]On the other hand, if we use classifier systems in a Michigan approach, then, strictly speaking it is not genetic programming anymore.

example. When evaluating a classifier system, all rules are checked (possibly in parallel) with *all* available input messages. The result of every classifier evaluation is added to the end result. This result is the output message list. For more details, we refer to [Gol89].

Input message list = { 001, 101, 110, 100 }

Condition	Action	Matches	Result
00# 101	111	yes	111
01# 1##	000	no	/
1## ~00#	###	no	/
1## ###	1#0	yes	100, 110

Output message list = { 111, 100, 110 }

Table 4.1: Illustration of how actions produce a result when the conditions match all messages in the input message list. ~ is negation of the next condition. A disjunction of two conditions is used for each classifier rule. The second rule does not match for input message 001. The third rule does not match because the negated condition is not satisfied for input message 001.

Using this representation for individuals of a genetic algorithm, we can easily introduce the necessary operators: cross-over is performed by selecting certain bits within the classifier expressions (genes) and exchanging them with the same genes from another individual. Mutation is easily implemented by selecting a number of bits and simply changing them to something else. The technique of using entire classifier systems as individuals within a genetic program is referred to as the Pittsburgh approach [Smi]. In comparison, the Michigan approach [LF93] treats every classifier- rule as an individual. After having introduced genetic algorithms and classifier systems we will now proceed with discussing reinforcement learning algorithms.

4.4 Reinforcement learning

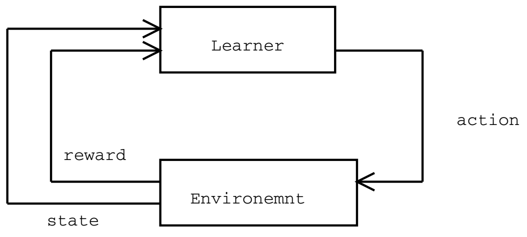


Figure 4.2: A reinforcement learner

THE APPROACH WE WILL USE to learn how to keep Petri-nets alive will be based on a reinforcement learning technique. Therefore, we will now introduce what a reinforcement learning technique is. Reinforcement learning [SAG98, KLAPM96] is originally an on-line technique where a learner tries to maximize the accumulated reward it receives from the environment. The actions the learner can take in a given situation are initially unknown, so the learner itself needs to discover a good way of working. The typical challenge for a reinforcement learning algorithm is that actions taken in a certain context will influence the future environment and rewards the learner will get.

Typical to the class of reinforcement learning algorithms is that it is defined based on the problem at hand. Reinforcement learning is defined as a solution to the problem in which a learner, which interacts with its environment, tries to reach a certain goal. Such a learner should be *emerged* in the environment, it should be able to *sense* the environment and it should be able to *act* upon the environment. The goal, or goals of such a problem should also be related to and expressible in the environment. A more formal definition of the problem in a more formal way will be given in section 4.4.4.

There are many kinds of reinforcement learning algorithms. For this dissertation, we will focus on one particular branch of algorithms, called $TD(\lambda)$-learners [Sut88, Tes92, SAG98]. The following sections will discuss this branch of algorithms in more detail. We will first introduce the elements most often encountered in reinforcement learners, second we will state the difference between episodic and continual tasks. Third, we will formally express the requirements of a reinforcement learning problem by means of Markov Decision Processes. Fourth, we will explore the value function and policy of a $TD(\lambda)$-learner. Fifth we will discuss the tradeoff between exploration and exploitation.

4.4.1 A Typical Reinforcement Learner

A typical reinforcement learning algorithm contains 4 elements:

1. A *policy,* which defines the actions the learner will take in a given context. This is typically what should be learned.

2. A *reward function* which defines the goal(s) of the learner as a numerical value. This function defines the *immediate* reward the learner gets from the environment. For instance, a learner searching for food gets a reward when it perceives food at its sensors.

3. A *value function,* which defines the actions that are good in the *long* run. Generally speaking, the *value* of a certain situation is the *maximum* reward the learner can expect to receive starting from that situation.

4. A *model* of the environment which represents what actions will turn one situation into another. In the original reinforcement approaches, models were avoided and only trial-and-error search was possible. However, some recent reinforcement learning algorithm approaches benefit from the use of models.

4.4.2 Episodic versus Continual tasks

In general, a reinforcement learning algorithm works on-line, i.e. emerged in the environment, without supervision. There is never an explicit training phase that teaches how the algorithm should behave in a real environment. However sometimes the same learner will be allowed multiple trials, in the form of episodes, to perform a task. An *episode* is the run of a learner until a terminal state is reached. Such a terminal state can be either a dead-state or a goal-state. After reaching such a state, a new episode starts, in which the same exact problem as before needs to be solved. However, during every episode the learner will be able to develop its behavior further until a suitable approach has been reached. Such a task is called an *episodic* task.

Other learning-tasks don't have terminal states and simply require the learner to maximize the total amount of reward it receives in the long run. Such a task is called a *continual* task.

4.4.3 Elements of a Reinforcement Learning Problem

Given the above general description of a reinforcement learning problem, we can now formally introduce the elements of a reinforcement learning algorithm. In chapter 9, we will use this formal approach to map the liveness problem to a reinforcement learning problem.

The action a learner takes at time t will be called a_t. The actions available to the learner in a situation s are denoted $A(s)$. When an action a_t is executed the resulting situation is denoted s_{t+1}. The reward received after executing a_t is a numerical value r_{t+1}. Time is considered to be discrete.

4.4.4 Markov Decision Processes

A reinforcement learner can never take all possible information into account when deciding upon its action, because either a) not all information is available, or b) it would require too much space to store all information. Therefore reinforcement learners require the environment to offer just enough information for the learner such that it can proceed. The situation-signal a learner receives should summarize in a compact way the past which has lead to this situation, such that no relevant information has been lost. If the state-signal succeeds in retaining this information it is said to have the *Markov*-property. For example, a checkers position, containing the position of all pieces on board, has the Markov-property, because it summarizes the past in a way such that no necessary information is lost for the future.

Formally, if the situation-signal has the Markov-property, the next situation and reward can be probabilistically expressed in terms of the current situation and reward. In the equation below $P_r$denotes a probability distribution.

$$P_r\{s_{t+1} = s', r_{t+1} = r'|s_t, a_t|\}$$

This notation denotes a chance that $s_{t+1} = s'$ and $r_{t+1} = r'$ if both s_t and a_t are given. In summary, a problem needs to have the Markov property before it can be considered to be a reinforcement learning problem.

4.4.5 The Value Function & Policy

We will now describe the value functions $V(s)$, $Q(s,t)$ and policy $\pi(s,a)$ as used by many reinforcement learners. A policy describes which action is favored in a given situation. To decide this, the policy uses a value-function, which describes the maximum possible reward a certain action could lead to. Typically, both the policy and the value function change over time. The policy balances the exploration and exploitation phases while the value function learns how good it is to be in a certain state.

Formally, given a certain situation, the policy decides the probability of an action to be chosen: $\pi_t(s,a)$ returns the probability that $a_t = a$ if $s_t = s$. The value function defines what the maximum future reward will be given a certain policy. However, because the time a learner might run can be infinite, the expected future reward may be also be infinite. Therefore, a discount factor is introduced which makes future rewards less interesting if they are placed further in the future.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \tag{4.1}$$

where $0 \leq \gamma < 1$.

Given this definition of a future reward, we can define the *state-value* function, which defines how good it is for a learner to be in a certain state. Of course, this depends on which actions the learner will take in certain situations, hence the policy π should be taken into account.

$$V^\pi(s) = E_\pi\{R_t|s_t = s\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\}$$

In the above equation, $E_\pi\{\}$ denotes the expected return value given that the learner follows policy π. Similarly, we can define the expected return the learner receives when it takes action a in state s.

$$Q^\pi(s,a) = E_\pi\{R_t|s_t = s, a_t = a\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\}$$

Q^π is called the *action-value* function under policy π.

Both the state-value as action-value functions cannot be calculated immediately because the expected future reward r_{t+k+1} is unknown. The goal of most reinforcement learning algorithms is to find approximations of these value-functions. In chapter 9 we will investigate how we can map the liveness problem (and our solution) to a function approximation of $Q(s, a)$.

- One such a technique estimates these functions by keeping track of all mean rewards following a certain state under a certain policy. This kind of estimation methods are called *Monte Carlo* methods. This technique however requires the existence of episodes, otherwise different action-decisions cannot be measured and the necessary averages of future rewards cannot be calculated. [AdFDJ03]
- If a perfect model of the environment is present, the value-functions are known and as such techniques such as *dynamic programming* can be used. However, they are computationally very heavy and most often there is no perfect model of the environment. [KLAPM96]
- Other estimation methods such as *Q-learning* [PW94] do a function approximation of the Q value-function. The technique used by this methods is called *temporal difference learning* because rewards are backwards propagated and over time the correct future rewards will be known. We will describe this technique in detail in section 9.3.2.

4.4.6 Exploration versus Exploitation

The difference between the policy and the value-function is that a value-function estimates what future reward could be possible, while the policy decides which action will effectively be taken. However, this action is not always necessarily the action that will lead to the highest reward. If the best action is taken, the policy is *exploiting* the value-function, otherwise it is *exploring* the environment. A tradeoff between exploitation and exploration must be made. If a learner exploits too much it will be blind to possibly better solutions which could result in higher rewards, while a learner which keeps on *exploring* the environment in a fully random way will end up accumulating very little reward.

An action selection method in which all information is exploited without ever trying new paths is called a *greedy* action selection strategy. The drawback of such a method is clearly that it gets easy stuck in sub-optimal solutions. A simple but effective variant of greedy action selection methods are the ϵ-action selection methods, in which most of the time the best available action is favored, but in ϵ % of the time an action is uniformly random selected from the set of possible actions. Eventually, all possible actions will be selected once by this method.

This wraps up our introduction of reinforcement learning.

4.5 Summary

IN THIS SECTION we have explained that we need an on-line adaptive learning algorithm that works on symbolic input and output. The algorithm needs to work on-line because of the nature of open distributed systems. The algorithm needs to work with symbolic input and output because of the nature of interfaces. The internal representation however can be either numerical or symbolic. As a result the only algorithm suitable to do the job is *reinforcement learning*, for which we explained the basic operation.

Second, in this chapter we introduced the use of genetic algorithms as a means to test the representation of a system, instead of an approach to solve a problem. If a certain problem can be solved easily under a certain representation, it is a good representation. If the same genetic algorithm has more trouble finding a solution the representation might not be so suitable. On the same representational track we also introduced classifier systems because we will investigate their usefulness chapter 11.

Third, we introduced the working of reinforcement learning algorithms. We explained the policy-function, value-function, reward function and the requirements posed upon a problem before it can be considered to be a reinforcement learning problem. One of these requirements was that the environment signal should summarize the past in such a way that no necessary information is lost. This property is called *Markov*. In chapter 9 we will need this property to ensure that the approach we use is indeed a valid reinforcement learning problem. The reinforcement learning technique introduced in this chapter will be used to create the liveness module.

We have now presented the preliminaries of this dissertation. In chapter 2 we explained the event based system we will use. In chapter 3 we introduced the use of Petri-nets as a means to formally specify an interface and in chapter 4 we introduced the learning algorithms we will need later on. Given these preliminaries we are now able to shift our attention to the case we will use to validate our thesis: concurrency strategy conflicts in open distributed systems.

Part II

The Case: Conflicting Concurrency Interfaces

Chapter 5

Introducing the Case: Concurrent Systems

OUR CASE FOCUSES ON THE CREATION of a concurrency adaptor. When working in an open distributed system, multiple components will require and provide all kinds of different locking strategies. If we want to interface with other components we need adaptors between the different synchronization approaches. The problem with this is that there are many possible synchronization approaches and that the number of combinations of different approaches is even larger. Hence we need an automatic concurrency adaptor. This chapter explains the concurrency problems we have in open distributed systems. We will gradually investigate a number of properties of concurrency guards in open distributed systems. This will enable us to identify a set of conflicts in chapter 6. The selection of these conflicts will be based on a number of variabilities of the different presented conflicts. Therefore we will present these at the end of this chapter.

5.1 Introduction

IN THIS CHAPTER WE INTRODUCE OUR CASE, which is concurrency. The motivation of choosing concurrency as a case, lies in the fact that we feel that concurrency is a problem that is often overlooked in open systems. As we will explain, every component that can be used by third party software, needs to offer a concurrency strategy. When multiple components offer different concurrency interfaces, conflicts might arise that have nothing to do with the core functionality of the component involved. This makes our case interesting because if we can mediate these conflicts automatically, we have actually removed a non-functional concern for developers.

For practical purposes the problem of concurrency in distributed systems will be simplified to its bare essentials. Instead of using real database servers or transaction servers and real clients, we will create our own mini-version of the problems present in these software systems. We do this for a number of reasons:

1. We want to offer an example of a client-server architecture, which covers all the essentials. So, our example should include concurrency problems, it should clearly illustrate the reason why one needs abstract techniques to manage concurrency and it should be able to express the notion of resources.

2. We want to minimize the core functionality of the program, because we this makes understanding all the different concurrency strategies more understandable.

3. To be able to use these concurrency strategies as a case we need to implement them under the form of components. Since it is hardly possible to find an existing combination of clients and servers which offer all kinds of different concurrency strategies but with the same core functionality, we had to create them ourselves.

The approach we use in this chapter to present concurrency is our own work. When appropriate we will point out papers that might be of interest to those who need more information. In this chapter we will focus on different concurrency strategies in such a way that it allows us to define a set of interesting conflicts in chapter 6.

5.2 Concurrency

THERE ARE TWO REASONS why distributed systems are concurrent systems. First, because they are running on multiple computers, where all those computers run in parallel and share resources. Secondly, because often distributed computations allow multiple users and have to support sessions. Concurrency might lead to a number of problems.

1. *Race conditions*: these occur when at least two sessions (directly or indirectly) read and write a shared variable in such a way that the order of the events determines the outcome of the application. The result of race conditions is that the application behaves indeterministically: depending on the order of execution different application states will occur. This indeterminism is hard to debug and all too often not wanted because it was not anticipated by the programmer. Often people tend to believe that race-conditions require a shared memory, however, this does not mean that race-conditions cannot occur in an event based system. For an example of this see section 2.7 on page 33.

2. *Deadlocks*: A group of components is deadlocked when there exists a closed cycle of components, each in turn waiting indefinitely for an event of the next component. Our definition is largely based on the notion of deadlocks within state-machines. When a Petri-net arrives in a situation where it still contains tokens, but no pre-condition holds for any transition, then it is in a deadlocked situation. (see 3.10 on page 65 or [Mur89, EN94]). In comparison to the operating system definitions of deadlocks, we find all necessary and sufficient conditions. A deadlock in operating system terminology requires [Bro97, Dic00]:

   (a) *mutual exclusion*, which means that only one process at a time can access a resource. In our event based system this is the case because a component can only serve *one* application session at a time. However, if the components themselves do offer a concurrency strategy that keeps track of session id's and allows multiple sessions to access the same resource then we might not have a deadlock. This is consistent with the Petri-net definition, because in such a case the Petri-net would have enabled transitions to support the new session.

   (b) *hold and wait*, which means that a session may hold some allocated resources while waiting for others. In our case this depends on what kind of concurrency strategy is offered by the involved components. However, also here we can determine whether a situation is a deadlock or not by looking at the involved Petri-net markings.

   (c) *no preemption*, which means that no resource can be taken away by force. In our definition we assume that no component can be taken away because this would make it impossible for the application to execute.

   (d) *circular wait*, which means that a closed chain of processes exists, such that each process holds a resources required by the next process in the chain. In our definition this is simply translated to the notion of components.

3. *Livelocks*: when measure of control is taken against race-conditions under the form of non-waiting locks, or when a measure of control is taken against deadlocks, by means of transactions, we can end up in a situation in which the system livelocks. In such a situation two processes start locking but encounter a problem halfway and release their locks again. They both restart again, again to release their resources after a while. In such a situation, the system is not waiting, on the contrary, it is working very hard, but it is not doing anything useful.

4. *Starvation*: an extra problem in concurrent systems is the problem of scheduling. By allowing processes to lock and unlock resources, some processes (or a group of processes) can (each in turn) lock a resource resulting in this resource being locked for a relatively long time thereby exceedingly slowing down other processes.

Understanding concurrency problems can be hard, solving them can be even harder, especially if we are faced with the possibility of partial failures.

5.3 The Whiteboard Case

AS A CASE THROUGHOUT THE THESIS we will use a whiteboard. A whiteboard is an application on which a number of different actors (users or other computer programs) can put and get elements. Normally a whiteboard is used as a support tool in group discussion systems. We will use it as a means to illustrate concurrency problems graphically. First we will discuss the whiteboard and its actors without any concurrency primitives. Afterward we will explain the problems and a number of standard solutions.

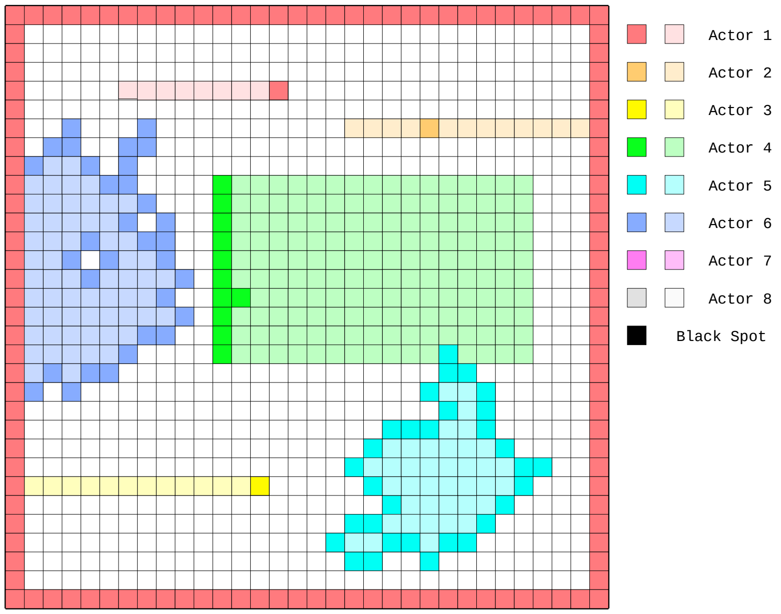


Figure 5.1: The whiteboard: every actor is represented by a color. Actor 1, 2 and 3 are moving dots. Actor 4 is a moving line and actor 5 and 6 are flood fills.

5.3.1 The Interface

The whiteboard has a very rudimentary interface:

- ⇝ `in JoinActor()` is an incoming message for the whiteboard. As a result **`out`** `ReturnJoinActor(<Integer|Result>)` is sent. The integer sent back to the client is the color that is assigned to that actor and can afterward be used to color squares on the board.

- ⇝ **`in`** `IsFree(<Pos|Pos>)` can be sent to check whether a certain position is empty or not. `Pos` is an x, and y coordinate. As a result **`out`** `ReturnIsFree(<Pos|Pos>, <Boolean|Result>)` is sent. When Result is *true* the position specified by `Pos` is free, otherwise it is not.

- ⇝ **in** `SetPosition(<Pos|Pos>,<Integer|Actor>)`: to set a certain position to the actor's color, send `SetPosition` to the whiteboard. When done the whiteboard will send **out** `ReturnSetPosition()` back to the requester. If the supplied integer 'actor' is zero, then the position is emptied. For illustrative purposes, when a position is cleared, a residue, in a lighter color, will remain: if an actor puts yellow on a certain position and then clears it that position will become lighter yellow and will be free for others to put something in. This way a trail can be seen where the actor has already been. See figure 5.1.
  out `RETURN_SetPosition()`: response of a SetPosition call.

The whole whiteboard itself is one component. It has one thread running and doesn't share data with other processes. Every actor on the whiteboard is also a component. There are 3 actors we will discuss: the moving dot actor; the moving line actor and the floodfill actor.

5.3.2 The Horizontally Moving Dot Actor

Algorithm 6 The horizontally moving dot algorithm

```
    $delta := (1,0)$
    $pos := (random(), random())$
    loop
       $next := pos + delta$
5:     if $IsFree(next)$ then
          $SetPosition(next, color)$
          $SetPosition(pos, 0)$
          $pos := next$
       else
10:       $delta := -delta$
```

The moving dot actor uses a simple algorithm to move over the whiteboard. It first checks whether the next position , left or right of the current position, is free. If it is it will mark that position and clear the old one. If the position is not free then it will change its direction. In figure 5.1, the red, orange and yellow actors are moving dots. Algorithm 6 covers the details. Please note that in all implementations we give we have to take care of the non blocking nature of the component system. In fact, we cannot wait until `IsFree` or `SetPosition` returns. However, for the sake of simplicity we omitted the original complex non-blocking code and replaced it with a more readable blocking version.

This simple actor *requires* and *provides* the following interface:

- ⇝ **o**`ut JoinActor()` when starting up the moving dot actor will join a whiteboard. After sending out this message the actor will passively wait for an
  in `RETURN_JoinActor(<Integer|Result>)`.

- ⇝ **out** `IsFree(<Pos|Pos>)` will be sent by the moving dot actor to check whether the next position to move to is free. The dot will wait until an
  in `RETURN_IsFree(<Pos|Pos>, <Boolean|Result>)` arrives.

- ⇝ **out** `SetPosition(<Pos|Pos>,<Integer|Actor>)` is sent out to set and clear the position (hence, move the dot). After every `SetPosition` the moving dot will wait until an **in** `RETURN_SetPosition()` arrives.

5.3.3 The Moving Line Actor

Aside from this simple actor, there is a more interesting figure that can be moved around the board: the moving line. It is a line of 10 pixels high, with a trailing dot behind it. The

Algorithm 7 The moving line algorithm

```
    $pos := (random(), random())$
    $delta := (1, 0)$
    $trail := (pos.x, 0)$
    $dtrail := (0, 1)$
 5: **loop**
      $newpos := pos + delta$
      $SetPosition(trail.x, pos.y + trail.y, 0)$
      $linefree := true$
      $i := 0$
10:   **while** $i <= 9$ and $linefree$ **do**
        $linefree := linefree$ and $IsFree(newpos.x, newpos.y + i)$
        $i := i + 1$
      **if** $linefree$ **then**
        $trail := trail + dtrail$
15:     **if** $trail.y = 9$ or $trail.y = 0$ **then**
          $dtrail := -dtrail$
        **for** $i := 0$ to $i = 9$ **do**
          $SetPosition(pos.x, pos.y + i, 0)$
        $trail.x := pos.x$
20:     $SetPosition(trail.x, pos.y + trail.y, actor)$
        **for** $i := 0$ to $i = 9$ **do**
          $SetPosition(newpos.x, newpos.y + i, color)$
        $pos := newpos$
      **else**
25:     $delta := -delta$
```

standard algorithm of the line (algorithm 7) checks whether the next line is free (lines 9,10,11 in the algorithm). If it is, the old line will be removed (lines 17,18), the trail will be drawn (line 19) and the next position of the line drawn (lines 20,21). The line's trail goes up and down relative to the origin of the line (lines 14,15,16). If the line bumps into something then it will change direction (line 24). Before checking whether the next line is free, the trail is removed, otherwise the line would not be able to turn around. (line 7). The interface required and provided by the line actor is exactly the same as the one for the moving dot actor.

5.3.4 The Floodfill Actor

The last actor we will discuss is the floodfill actor. This actor tries to fill the whiteboard by enlarging its own domain. The standard algorithm (algorithm 8) keeps a set of points that are owned by the flood actor, a set of border points and a set of seed points. A border point is a point owned by the floodfill actor, but with not all neighbors owned by the actor. A seed point is a set of points that are possible candidates to fill. They are not yet owned by the actor. Normally a point starts in the seed set. If the point is free, the point becomes a border and all 4 neighbor points are added to the seed set. If a point is solely surrounded by points owned by the actor then it becomes an owned point. Points that became owned are cleared on the whiteboard but remain owned.This can be seen in figure 5.1. In the figure the purple and turquoise actor are flood actors. The interface required and provided by the flood actor is exactly the same as the one for the moving dot actor.

Algorithm 8 The flood actor algorithm

```
$ismine := \{\}$
$seeds := \{(random(), random())\}$
$border := \{\}$
**while** $\#seeds > 0$ **do**
  $pos := seeds.take(random())$
  **if** $pos \notin ismine$ and $IsFree(pos)$ **then**
    $SetPosition(pos, color)$
    $ismine := ismine \cup \{pos\}$
    $border := border \cup \{pos\}$
    $seeds := seeds \cup \{pos.left(), pos.right(), pos.up(), pos.down()\}$
  **for** each $\epsilon \in border$ **do**
    **if** $\{\epsilon.left(), \epsilon.right(), \epsilon.up(), \epsilon.down()\} \subset ismine$ **then**
      $border := border \setminus \{\epsilon\}$
      $ismine := ismine \cup \{\epsilon\}$
      $SetPosition(\epsilon, 0)$
```

5.4 Race Conditions

WHEN WE LOOK AT THE INTENDED BEHAVIOR of all actors, we see that it is of vital importance that no actor crosses the boundaries of another. Dots should not go through each other, lines should bump, dots should not enter floodfill actors and floodfill actors should not cross each other either. Furthermore a line should always have a trail that goes up and down relative to the origin of the line. In this section we will illustrate that none of these requirements are met if we don't use any concurrency primitives. Let's investigate some of the problems

5.4.1 A Selection of Race Conditions

- ⇝ *Moving dots can go through each other*: Figure 5.2 illustrates how two dots (as specified in algorithm 6) can pass each other without bumping. The problem occurs when both dots detect that a certain position is free and then both take that position. Afterward both dots will simply continue their way and since no other dot forms an obstacle, they can proceed. The netto result is that they did not collide, a thing they are required to do.
- ⇝ *Dots can enter a floodfill*: As stated in algorithm 8, the floodfill actor will first enlarge its set of possible seeds, in the next iteration a seed will be taken and checked whether it is free, if it is free the seed becomes ours. It is in this last step that a concurrency problem can occur: between checking whether the position is free and actually taking it. Figure 5.3 illustrates this.
- ⇝ In the same way *line actors can pass each other* and *floodfill actors can enter each other*: They will check whether a position is free and then acquire that position, blind to the fact that another actor has already occupied that position.

5.4.2 Different Solutions towards Race Conditions

The behavior described above is typically called a race condition. It means that two (or more) concurrent processes try to get to the same resources and are actually 'racing' to be the first. To deal with this kind of problems, a number of alternate tracks exist.

1. *Detecting concurrency problems*: a big problem of concurrency problems is that they are very scheduler dependent. We can work for years with the same code and after changing the scheduling behavior of the kernel we notice how things start to fail sometimes. At the

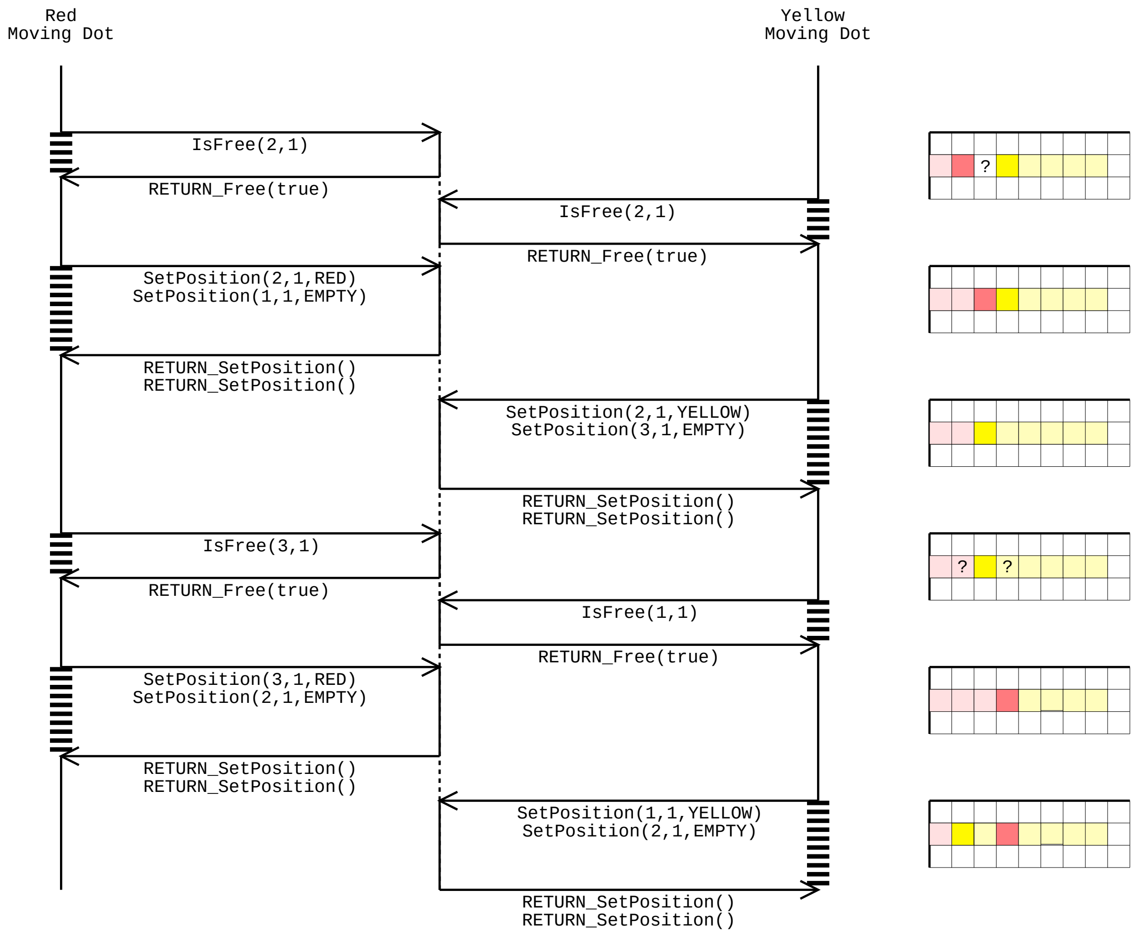


Figure 5.2: How two moving dots can pass each other without bumping.

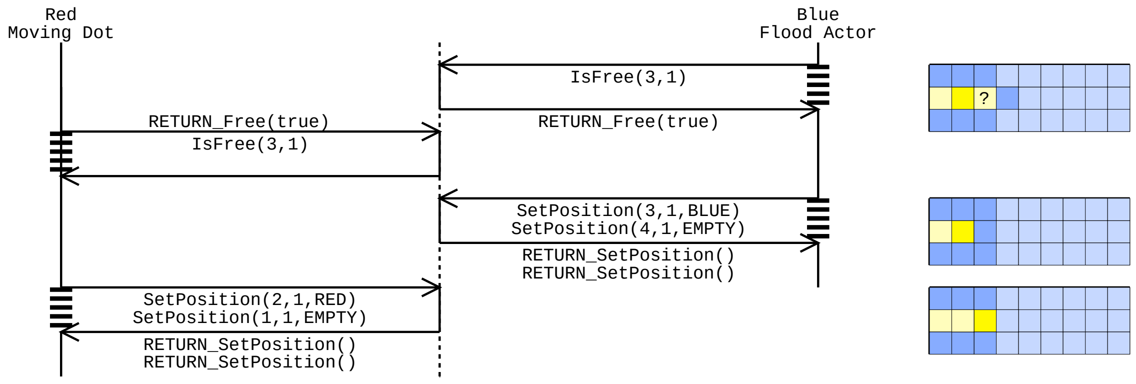


Figure 5.3: A moving dot crossing the flood actors boundaries

moment we see these spurious errors we probably want to reproduce the bug and start debugging. The problem here is that under a debugger the scheduling behavior of programs tends to be different and concurrency problems don't show up anymore. This is what is called an *Heisenbug*. There is work done to help people debug such systems by means of record and replay. During the record phase a debugger runs the program and simply remembers the order in which locks or acquired or released. Afterward the debugger can replay the original execution, with the same order of events [RB02, Gar97]. Within thread based system, *Lamport* clocks [Lam77] are typically used for this. Within event based systems we can simply record all events. [CL94] discusses how such a the ordering of events within a message passing system (such as the one we are using) can be implemented.

2. *Formal verification*: What we often want to do is to check programs for concurrency problems in a more formal way. Therefore, something often done to describe and detect concurrency problems is to specify pre- and post-conditions. In our case, for the moving dot actor, a possible precondition could be: the next position is free and the current position is mine. The postcondition after moving would then be: the next position is mine and the current position is free. It is clear that such a pre- and postconditions can be used to detect race conditions at runtime. If we want to check the possibility of race-conditions statically, formal techniques exist that will automatically deduce how race conditions can be created. [CG]

The problem with *open* distributed systems is that we cannot verify those pre- and postconditions if we don't have all participating actors at hand. Now, let's see how concurrency problems can be solved in open distributed systems.

5.5 Centralized Atomic Operations

NONE OF THE BEFORE MENTIONED RACE CONDITIONS would exists if we modify our server a little bit. Instead of offering only an `IsFree` and `SetPosition` operator, we could add an extra operator to the whiteboard:

⇝ **in** `MoveWhenFree(<Position>, <Direction>)`: Position is the position of the point to move. Direction can be either left, right, up or down. This operator will check whether the point we're moving to is free. If it is it will swap its content with position.After doing so, a message **out** `RetrunMoveWhenFree(true)` will be sent. When the position was not free a message **out** `ReturnMoveWhenFree(false)` is sent. This operator will also return false (meaning not free), when the target position is used by the same color as the original requester.

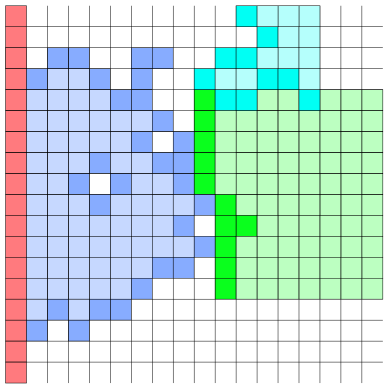


Figure 5.4: Moving a line with the `MoveWhenFree` operator.

For most actors (the moving dot actor, the line actor and possible others), this solution might work, however this solution cannot guarantee that every possible use of the component an be expressed in such a way that no race-conditions occur. For instance, with this operation, it would be very difficult to make the floodfill actor to work correctly, without race-conditions. The biggest problem is that the floodfill actor in essence doesn't move, instead it takes new points if they border to the actor. As such implementing a floodfill actor in a safe way would mean that we can only move dots away from a position that is bordered by three other flood-dots. This would require an initial seed of at least 5 positions and would require additional code to input new blocks in the interior of the floodfill.

This solution illustrates that extending a server to support every possible critical section in one message is no solution at all, especially not in *open* distributed systems. Instead, it is a very local solution only to solve the problem of specific clients. Therefore we need some better solution. We cannot for every possible critical section end up modifying the server.

5.6 Non-Waiting Atomic Operations & Starvation

IN OPEN DISTRIBUTED SYSTEMS we *need* some form of critical sections, otherwise different actors can change the internal state of a component, without taking into account other components. Placing all possible critical sections at the server is not good enough because this would not allow unanticipated behavior for other components. Therefore we need a more abstract way to specify our critical sections. In fact, we need to specify the beginning and the ending of a critical section in such a way that no component allows access from another component unless it has obtained an atomic operation id. In such a scenario every component would provide a certain synchronization interface, for instance, defined as:

```
in enter(<Integer|Id>)
out enter_ok(<Integer|Id>)
out enter_fail(<Integer|Id>)
in leave(<Integer|Id>)
out leave_ok(<Integer|Id>)
out leave_fail(<Integer|Id>)
```

The implementation of the component offering such an interface of course requires some changes. to ensure that the requester can execute operations upon a server component, every incoming message id should be verified. This should be checked for every action, but this can be done quite easily.

How the semantics are defined is a bit more difficult. Two orthogonal features need to be investigated. First there is the question whether those atomic operations are reentrant or not, second there is the problem whether an atomic operation waits or not when entering. In total this gives about 4 essentially different synchronization interfaces. We will explain two of them in more detail below. The other two are reserved for the next section. We will define the semantics of the different approaches by means of pseudo code.

5.6.1 Non-Reentrant Synchronization Semantics

Non-waiting, non reentrant enter/leave synchronization semantics. (Algorithm 9). This way of locking allows the interface requester to start an atomic operation on the interface provider by *entering* it. The `enter` takes an argument that should be a global unique identifier. In return the interface provider will return `EnterOk` when the atomic operation could be started, `Enter-Fail` otherwise. At the moment a component has an atomic operation initiated on the server it cannot re-enter the server, not even from within the same atomic operation. This is called non re-entrance.

Algorithm 9 non-waiting, non reentrant, enter/leave locking semantics

```
$lockedby := empty$

incoming $Enter$(<Id>)
  **if** $lockedby = empty$ **then**
    $lockedby$ =<Id>
    $requester$ .. $EnterOk$(<Id>)
  **else**
    $requester$ .. $EnterFail$(<Id>)

incoming $Leave$(<id>)
  **if** $lockedby$ =<id> **then**
    $lockedby := empty$
    $requester$ .. $LeaveOk$(<Id>)
  **else**
    $requester$ .. $LeaveFail$(<Id>)
```

5.6.2 Reentrant Synchronization Semantics

Algorithm 10 non-waiting, reentrant enter/leave locking semantics

```
$lockedby := empty$
$lockcount := 0$

incoming $Enter$(<Id>)
  **if** $lockedby = empty$ or $lockedby$ = <Id> **then**
    $lockedby$ :=<Id>
    $lockcount := lockcount + 1$
    $requester$ .. $EnterOk$(<Id>)
  **else**
    $requester$ .. $EnterFail$(<Id>)

incoming $Leave$(<Id>)
  **if** $lockedby$ = <Id> **then**
    $requester$ .. $LeaveOk$(<Id>)
    $lockcount := lockcount - 1$
    **if** $lockcount = 0$ **then**
      $lockedby := empty$
  **else**
    $requester$ .. $LeaveFail$(<Id>)
```

Non-waiting, reentrant enter/leave synchronization semantics. (Algorithm 10). When such an interface is provided, the same interface client can start multiple nested atomic operations on the server. This is especially useful when working with recursive functions. The algorithm itself simply keeps a lock counter that is increased every time a client enters. When a client leaves an atomic operation, the lock counter is decreased and `LeaveOk` is sent back. At the moment the lock counter reaches zero `lockedby` is emptied and another actor can enter an atomic operation. Currently the reentrant semantics return `LeaveOk` when an atomic operation is ended. This is not necessarily always the case. It would also make sense to differentiate here. For example, return a `LeaveNested` when there are still other atomic operations running and a `LeaveOk` when the last atomic operation has ended. In the same way, the non-reentrant semantics could be changed to return `EnteredAlready` when such an atomic operation has already been started.

5.6.3 Discussion

The two algorithms presented here, require the client to supply a *globally unique identifier* to the server. This is clearly not a realistic requirement. Normally the server will choose a number of its own and return that as a unique identifier. In section 5.10 we will present more about this.

The two above algorithms are both non-waiting algorithms, this means that if the interface provider returns an `EnterFail` message, then the interface requester will need to retry again at a later time. This is a source of trouble, because, chances are high that the interface requester will immediately try to enter again until he finally could start his atomic operation. This leads us to the following problems:

- ⇝ The more components are accessing the server, the more useless network traffic will take place. This network traffic would mainly consists of `enter` requests and `EnterFails`.
- ⇝ Moreover, since there is no fairness involved, the client flooding the server the most will probably obtain the server lock first. Clients with a slow network connection, located at a distance will have no fair chance to lock the server.

5.7 Waiting Atomic Operations & Deadlocks

To avoid network congestion and starvation of distant components, we will now investigate how the earlier defined API can be implemented with waiting semantics. We will again distinguish between reentrant and non-reentrant semantics.

5.7.1 Non-Reentrant Synchronization Semantics

Algorithm 11 Waiting, non reentrant enter/leave locking semantics

```
$lockedby := empty$
$waitinglist$ :=new queue()
incoming $Enter$(<Id>)
  **if** $lockedby = empty$ **then**
    $lockedby$ :=<Id>
    $requester .. EnterOk$(<Id>)
  **else**
    $waitinglist.push(message)$
incoming $Leave$(<Id>)
  **if** $lockedby$ = <Id> **then**
    $requester .. LeaveOk$(<Id>)
    $lockedby := empty$
    **if** $!waitinglist.empty()$ **then**
      $Enter(waitinglist.pop())$
  **else**
    $requester .. LeaveFail$(<Id>)
```

Waiting, non reentrant enter/leave locking semantics: algorithm 11). These semantics are the same as the non-waiting locking semantics. The biggest difference is when somebody wants to enter an atomic operation on the interface provider. At the moment there is already an atomic operation running, the requester will be placed in a queue. As such, an `EnterFail` is never returned, instead the possible `EnterOk` is held back for a later time. At the moment a leave request arrives with the correct id, the atomic operation is finished and the first in the waiting list is informed and can start with his set of atomic operations.

5.7.2 Reentrant Synchronization Semantics

Algorithm 12 Waiting, reentrant enter/leave locking semantics

```
$lockedby := empty$
$lockcount := 0$
$waitinglist$ :=new queue()
incoming $Enter$(<Id>)
  if $lockedby = empty$ or $lockedby =$ <Id> then
    $lockedby$ :=<Id>
    $lockcount := lockcount + 1$
    $requester$ .. $EnterOk$(<Id>)
  else
    $waitinglist.pushlast(message)$
incoming $Leave$(<Id>)
  if $lockedby =$ <Id> then
    $requester$ .. $LeaveOk$(<Id>)
    $lockcount := lockcount - 1$
    if $lockcount = 0$ then
      $lockedby := empty$
      if $!waitinglist.empty()$ then
        $enter(waitinglist.popfirst())$
  else
    $requester$ .. $LeaveFail$(<Id>)
```

Waiting, reentrant enter/leave synchronization semantics. (Algorithm 12). The reentrant version has a similar protocol behavior. The only difference is that this algorithm will increase and decrease a lock counter in response to `Enter` and `Leave` respectively. At the moment the lock counter reaches zero, the next waiting request is considered. Some variations could exists upon these semantics.

5.7.3 Discussion

The non-waiting atomic operations had the problem of starvation and network flooding. In the same way, waiting locking semantics have their own problems: *deadlocks*. A deadlock is technically speaking a situation in which multiple processes are waiting for each other to do something. So they are all virtually dead. There are two interesting situations.

⇝ A typical deadlock with the non-reentrant locking strategy is a deadlock within one component. If a client component tries to lock the server component *twice*, due to some nested function, it will simply stop and place the second component in the queue. Of course since the client component cannot proceed, because it still doesn't have the lock, it keeps on waiting, thereby never releasing the first atomic operation started by it. The result is a deadlock, due to non re-entrance.

⇝ Another deadlock situation happens when two components try to lock each other (due to some external request). This is pictured in figure 5.5. Component A starts a green atomic operation on component B. Component B accepts this. Component D starts a purple atomic operation on component C. Component C accepts this. Component D wants to act on component C, as a result component C needs to lock component B. This is impossible, thus component C is placed in wait. Component A now requests component B to act. Component B needs to start an operation on component C. This is also impossible, so now component B is also placed in wait. As a result all components stop working.

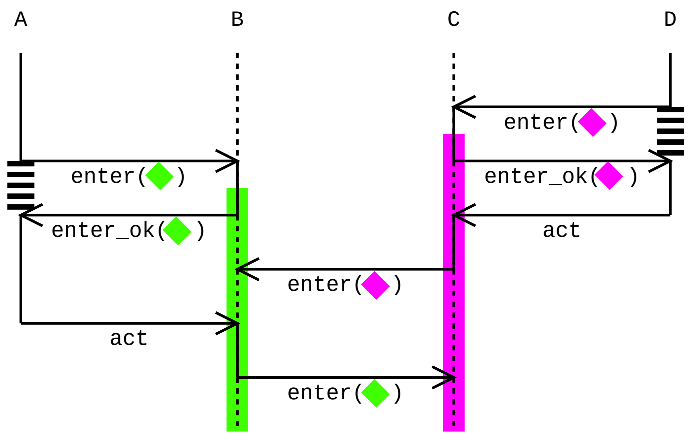


Figure 5.5: Mutual exclusion of two components. All components offer a waiting locking strategy.

Aside from the fact that deadlocks stop components from doing anything useful, there are some extra problems involved. Since some components simply stop working within a deadlock, other components that are dependent on those deadlocked components will eventually also cease to work. In our small example, component D and component A are in such a situation. Eventually, a whole application may end up deadlocked.

5.7.4 Different Solutions to Deadlocks

There are 3 different approaches to solve the problem of deadlocks:

⇝ *detection* of deadlocks: this is a technique in which deadlocks are detected at runtime and then broken by some means. Typically a detected deadlock is solved by removing one or all of the offending processes. This track will not be investigated further in this dissertation because it effectively renders the application useless. Another technique is rolling back the entire state of one (or all) of the involved components to an older state and restarting them from there. In our situation this is not possible because it is very difficult to roll back a distributed state to a previous moment in time. Also the possibility of extra user input might complete this scheme.

⇝ *avoidance* of deadlocks: this technique requires the ability to know the future locking behavior of the involved components. This knowledge can dynamically lead to a correct decision to avoid deadlocks. However, this information is difficult to obtain if the components are not written to specify which resources they will lock in the future.

⇝ *prevention* of deadlocks: this technique alters the variables of the surroundings in such a way that deadlocks doesn't occur. Different approaches are possible

  – instead of waiting for locks, a component can return a `lock_false` message. By doing so, deadlocks will no longer occur. We have already discussed this approach in section 5.6. In the next section 5.8 we will discuss some problems involved with this technique.
  – define a linear ordering on the locking requests such that circular waits are no longer possible. We will discuss this in more detail in section 5.8.3.

5.8 Locking Resources & Livelocks

IN THE ABOVE MENTIONED SYNCHRONIZATION SOLUTIONS, we specify the start of an atomic operation on the server with some form of message. Once an atomic operation is started no other actors can start an atomic operation. This is not always as efficient as it could be. Consider our whiteboard example. If a client wants to set the color of two squares, it has to start an atomic operation on the whole server. This is in essence the same as locking the full server. Suppose now that a second actor also wants to draw some pixels but on the opposite side of the whiteboard. It is clear that those two operations *can* coincide, but won't because both need a server lock.

5.8.1 Granular Operations: Locks

A solution to this problem is to introduce a lock for every square on the whiteboard. In doing so a client actor can request the server to lock a number of squares, which it can access afterward, but still concurrently with other actors that have locked other positions on the board. The API for such a lock is extended somewhat. Instead of declaring where we want to enter we specify what position to lock (or unlock). The semantics can be implemented in the same way as specified before. We can have a combination of a waiting/non-waiting strategy with a reentrant/non-reentrant locking strategy. With respect to terminology, such locks are typically called semaphores. A reentrant lock is sometimes called a counting semaphore, while a non reentrant lock is often called a binary semaphore. [Lea00]

We introduced a lock for each square on the whiteboard. We could also introduce a lock per x-line or a lock per y-line on the field, or a lock per 4 squares on the field. We choose to map our resources to squares.

Algorithm 13 The moving dot algorithm whereby a waiting lock protocol is expected from the server.

```
$delta := (1, 0)$
$pos := (random(), random())$
**loop**
   $next := pos + delta$
   $Lock(next)$
   $Lock(pos)$
   **if** $IsFree(next)$ **then**
      $SetPosition(next, color)$
      $SetPosition(pos, 0)$
      $Unlock(pos)$
      $pos := next$
   **else**
      $delta := -delta$
      $Unlock(pos)$
   $Unlock(next)$
```

Using a waiting non reentrant locking strategy to solve the concurrency problems when moving dots, flooding the whiteboard or moving lines with their trail is easy. Algorithm 13 shows how this can be done. Before actually checking whether a position is free, we lock the current position as well as the next position. If the next is free, we move and release both locks. If the position is not free, we also release both locks but we turn around. The same thing can be done for the moving line actor and the floodfill actor.

5.8.2 Problems of Fine Grained Locks: Livelocks

With this solution there still exists a plethora of problems:

⇝ When using a waiting lock strategy we have the problem of deadlocks. See figure 5.6. At the moment the yellow actor wants to lock its area, that is the current position and the next position, it will first try to obtain a lock at the next position (2,1) and then try to lock its own position. The problem arises when the red actor now has already locked its next position

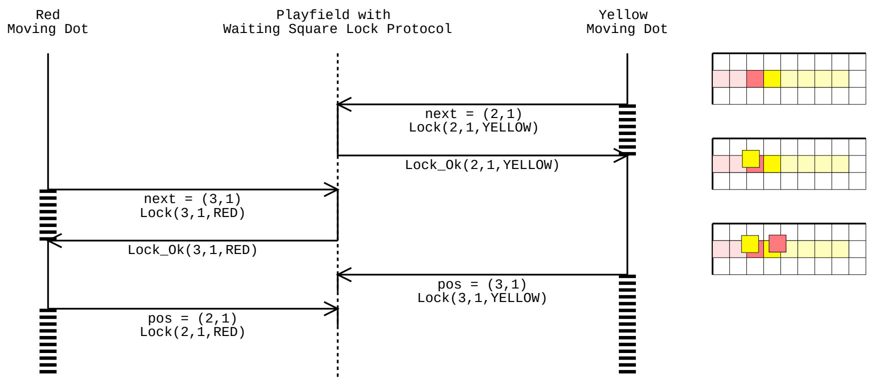


Figure 5.6: Mutual exclusion between two actors trying to lock the same area of the whiteboard. A square layered above another square denotes a lock. So the yellow square above the Red square means that the position is colored red, but is locked by yellow.

(3,1), our current position. In this situation both dots become deadlocked because they are both waiting to acquire the same lock, a lock that will stay locked until one actor gets the locks it is waiting for.

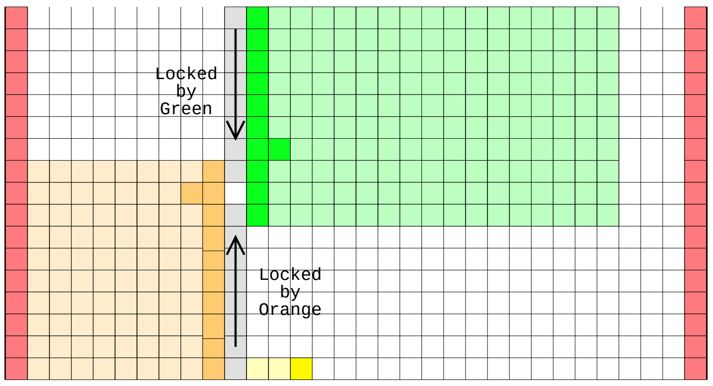


Figure 5.7: Illustration of a livelock. The acquiring of locks for green and orange will continue until one finally succeeds in getting a whole line locked. This can take some time.

⇝ When using a non-waiting locking protocol the normal behavior of the client is to try to lock a certain position and if it doesn't succeed release all the locks acquired yet. This can result in a *livelock* (see figure 5.7): a situation in which a number of clients try to lock the same position over and over again and stumble every time over the same problem. In our example this can be best visualized by using the line actor. Suppose that the line actor locks all its squares from top to bottom when going left, but from bottom to top when going right. At the moment both lines encounter each other at the same x-coordinate, then at a certain moment they both will need to acquire a position, common to both. Both will release all their locks and retry. This is called a *livelock* and results in starvation of both processes. The larger the overlapping part between the lines the longer this livelock will persist, simply because one of both lines needs to keep away from the shared part during the entire locking operation of the other actor. The larger the shared part, the more unlikely this becomes. In other terms: the more shared resources involved, the longer a livelock will persist, unless an absolutely unfair scheduling system is introduced.

5.8.3 Solutions to Livelocks

Looking at these problems, we see that there are a number of possible solutions when working with non nested locks.

A typical solution to this kind of problems is called *staircasing*. Here locks should be acquired in a certain order. In our example it would make sense to sort all squares from left to right and top to bottom. When doing so, deadlocks are impossible. When a lock is requested and cannot be assigned we are sure that the other actor can continue because it already has all the locks it needs with an order smaller than the one it is requesting. As such, we don't need to unlock any of our already acquired locks. When using staircasing it is important to acquire the locks *in order*.

Algorithm 14 How locks can be acquired within one atomic operation on the server.

```
$delta := (1, 0)$
$pos := (random(), random())$
**loop**
    $next := pos + delta$
    $StartLocking()$
    $Lock(next)$
    $Lock(pos)$
    $StopLocking()$
    **if** $IsFree(next)$ **then**
        ...
```

Another possible solution is to make locking and unlocking itself an atomic operation, so extending the lock and unlock operations with *a set of locks* to acquire or release. (See algorithm 14). The two green lines start and stop the 'atomic locking operation'. Within this atomic operation two locks are acquired. Please note that there is no need to release the locks within a critical section. This way we are sure that we have all the locks we need at once or none at all.

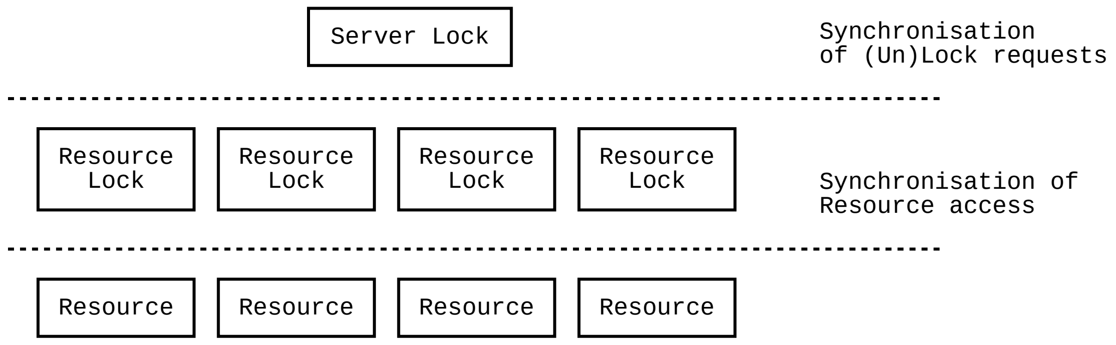


Figure 5.8: Synchronization Layers

In fact we are now solving a synchronization problem between locks, and no longer between resources. We are specifying locks themselves as resources where the access to the resource (the acquiring of locks) should be guarded by another lock. This layering is pictured in figure 5.8.

Using the above techniques (staircasing or making locking an atomic operation by itself) is difficult when the programmer expects nested locking semantics. Both techniques rely on the fact that at a certain point in time the program knows all the locks it will eventually need. When working with subroutines (or subprograms) that autonomously acquire locks this can be very difficult. One workaround is first to ask all subroutines what they will eventually need, collect all those locks and then acquire them within one operation.

However, if it is impractical to require that a process specifies all the resources it needs at once, we might need to resort to transactions, which we will explain in the next section.

5.9 Transactions & Partial Failure

A TRANSACTION IS AN ABSTRACT ATOMIC operation within which changes can be made on a component. When the transaction is finished it can either commit or abort. Committing a transaction means that all the changes made within its critical section become true and visible to other participants. Aborting a transaction means that all the changes made within the critical section are undone.

Transactions are necessary in systems where we cannot order locks because either there is no obvious order, because we cannot force actors to respect that order, or because an actor cannot specify all it needs at once. Another situation where we need transactions is when we think about failures. What will happen when one of the actors on the whiteboard has locked certain positions and then dies. Can the server recover from such a situation ? If the server supports transactions and it can detect client-death (by means of a time-out, or by detecting a broken socket) it can rollback the transactions owned by that client.

Algorithm 15 Semantics of a rudimentary non-waiting, non reentrant transaction system.

```
  $locks := \{\}$    $//contains(transaction, resource, state) - tuples$
  incoming $BeginTransaction()$
    $requester .. transaction(uniquetxid)$
  incoming $CommitTransaction(tx)$
    $locks := locks \setminus \{lock \in locks \mid lock.tx = tx\}$
  incoming $AbortTransaction(tx)$
    for all $lock \in locks \mid lock.tx = tx$ do
      $lock.res$.setstate($lock.state$)
    $CommitTransaction(tx)$
  incoming $lock(tx, res)$
    if $\exists\ lock \in locks \mid lock.tx = tx$ and $lock.res = res$ then
      $requester .. lock\_true()$
    else if $\exists\ lock \in locks \mid lock.tx \neq tx$ and $lock.res = res$ then
      $requester .. lock\_false()$
    else
      $locks := locks \cup \{(tx, res, res.$getState()$)\}$
```

Transactions can allow the nesting of operations. When, within one transaction another is started and the locks required to execute the inner transaction cannot be obtained, the outer transaction might abort also. From the programmer's point of view this is what we want, also from the point of view from the server transactions are good because the server will always reside in a valid state, something that cannot be guaranteed without transactions.

The problem of livelocks still remains of course. Fortunately, this can be remedied at the server side. The problem is no longer solely dependent on the network traffic or the behavior of the clients, but on the scheduling behavior of the server. For instance a server may decide to set a lock request in wait until the lock can be obtained, or it can return a lock-failure.

Depending on the context within which people talk about transactions there can be a difference between read-locks, write-locks, and the way locks are treated. If smaller locks are defined then there is a greater flexibility to optimize the concurrent access to resources. Similar to the granularity of resources, we can consider a lock to be a resource on its own.

Implementing a simple transaction system can be done by keeping in memory which locks belongs to which transaction. Once a lock is released the lock keeps on belonging to the original transaction. Only, when the transaction commits, all its locks are released. When the transaction aborts, all its locks are released and all changes to the resources are rolled back. Algorithm 15 illustrates the rudimentary semantics of a non-waiting, non recursive transaction synchronization interface. Such transactions can be easily used from a programmers point of view. We illustrate this by making the line actor movable (Figure 16). The reader interested in transactions can read [OHE96, CDK94].

Algorithm 16 The moving line algorithm using a non-waiting transaction interface.

```
$pos := (random(), random())$
$delta := (1, 0)$
$trail := (pos.x, 0)$
$dtrail := (0, 1)$
loop
    $tx := 0$
    retry: if $tx$ then $abortTransaction(tx)$
    $tx := beginTransaction()$
    $newpos := pos + delta$
    if $!lock(tx, (trail.x, pos.y + trail.y))$ then
        goto retry
    $SetPosition((trail.x, pos.y + trail.y), 0)$
    $linefree := true$
    $i := 0$
    while $i <= 9$ and $linefree$ do
        if $!lock(tx, (newpos.x, newpos.y + i))$ then
            goto retry
        $linefree := linefree$ and $IsFree((newpos.x, newpos.y + i))$
        $i := i + 1$
    if $linefree$ then
        for $i := 0$ to $i = 9$ do
            if $!lock(tx, (pos.x, pos.y + i))$ then
                goto retry
            $SetPosition((pos.x, pos.y + i), 0)$
        $trail := trail + dtrail$
        if $trail.y = 9$ or $trail.y = 0$ then
            $dtrail := -dtrail$
        $trail.x := pos.x$
        if $!lock(tx, (trail.x, pos.y + trail.y))$ then
            goto retry
        $SetPosition((trail.x, pos.y + trail.y), actor)$
        for $i := 0$ to $i = 9$ do
            $SetPosition((newpos.x, newpos.y + i), color)$
        $pos := newpos$
    else
        $delta := -delta$
    $commitTransaction(tx)$
```

5.10 Peer to Peer Concurrency & Distributed Transactions

WITHIN OPEN PEER TO PEER NETWORKS concurrency control and guarding is fairly difficult, because in a peer to peer system several components together provide a certain global behavior. So if one wants to take this global behavior from one correct state to another correct state, one needs to take all participating components *within one operation* to this new state.

To do so we need a transaction that spans multiple components and can commit all components or abort all components. This is called a distributed transaction. Normally distributed transactions are provided by one server that contacts the necessary components and commits or rolls back transactions as necessary.

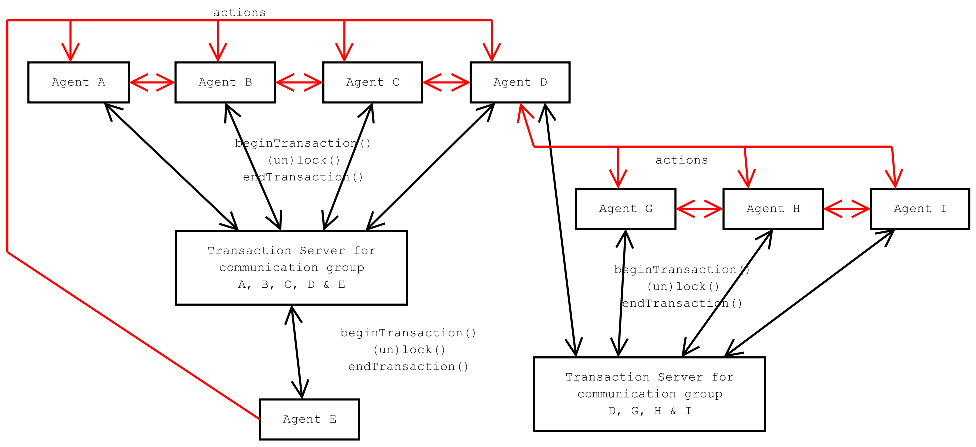


Figure 5.9: Distributed Transaction server.

In figure 5.9 we see how component *E* needs to go through a transaction server before it can start certain actions on a certain set of components. The transaction server takes care of transmitting all incoming messages to the other communication partners, possible mapping different transaction id's to the same number. In essence, all locking logic should go through this transaction server. The transaction server is available per group resources needed per client, which means that all components taking part in a certain session need to go through the same transaction server. So in fact we added another extra layer to solve concurrency problems between multiple components. (Pictured in figure 5.10).

The transaction manager per group is not difficult to conceive. It needs to access the transaction ports on all participating components and needs to have a mapping. This mapping ensures that when a `beginTransaction` comes in, the transactions started on all participating components will be represented by one transaction id. When a lock request comes in, the supplied transaction id should for every component be mapped to the correct effective transaction id. This constitutes a problem because a component is normally not aware of some sort of transaction manager. This means that all components need to agree to use the specified session transaction manager.

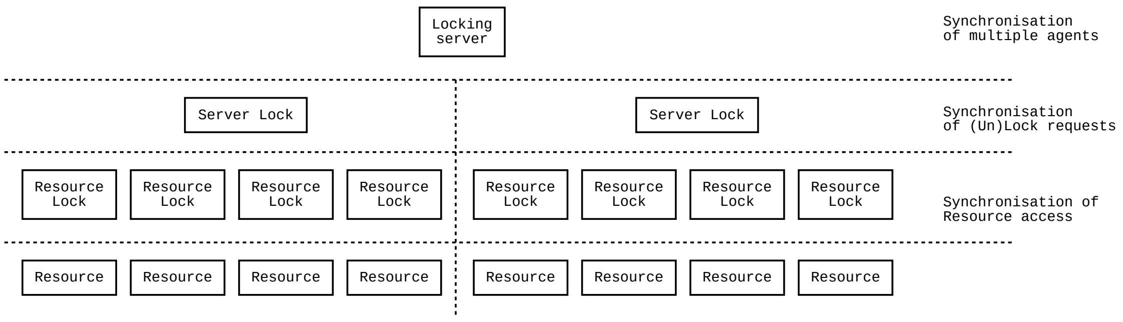

Locking server
Synchronisation of multiple agents
Server Lock
Server Lock
Synchronisation of (Un)Lock requests
Resource Lock
Synchronisation of Resource access
Resource

Figure 5.10: Synchronization layers within peer to peer systems

Of course, there are problems with this setup. Other concurrency strategies worked well because their behavior was unambiguous. A transaction was either committed or not, a lock was obtained or not and so on. The problem we have here is that with a distributed transaction we can have transactions that are partially committed. We can have a situation where the transaction server starts committing all its transactions but fails to commit the last transaction because the involved component has died. In such a situation the resulting global state is invalid, even when the component recovers its old state. There is not much that we can do about this, except introducing extra redundancy. Distributed transactions are discussed in [CDK94].

5.11 Commonalities & Variabilities

WITHIN THE PREVIOUS SECTIONS we have introduced a number of concurrency strategies. All these different concurrency strategies can be described by a set of commonalities and variabilities. The common issue is the fact that we are talking about resources and about critical sections: how can we ensure a valid state transition ? Below we will present a set of variabilities which can define a concurrency strategy.

1. *Syntactical*: how a component calls another component, with which parameters, with what kind of symbols and names. Syntax as a common term refers to the structural aspects of a language. In our case we will simply stick to the symbols and data structures at hand.

2. *Control flow*: in what sequence do we need to send messages ? Will requests wait until they can return, or will they return something like 'try again later'. We have seen two examples of this: the waiting locking strategy and non-waiting locking strategy

3. *Re-entrance*: can the same lock be obtained multiple times or not. If it can, such as in the counting semaphore, the locking strategy is reentrant, in the other case, such as the binary semaphore, the locking strategy is non-reentrant.

4. *Resources*: what are the resources we are talking about, and more specifically, what is their granularity ? Can we only lock the complete whiteboard, can we lock lines or can we lock individual squares ?

5. *Transition*: how is time defined. This is important with respect to the state transitions. Is a state transition always in effect immediately, or is the transition effective after committing a transaction. If so, can we go back in time (roll back) and can we go forward in time (roll forward after recovery)

6. *Layering*: most of the time multiple basic synchronization mechanisms are layered, how this is done is another variability.

The above variabilities can be used to describe a concurrent strategy. We must now investigate how they relate to each other, which variabilities can be modified without impact on other variabilities and which variabilities are influenced when another variability is shifted. If two variabilities do not correlate they are said to be orthogonal.

Control flow can be modified without any immediate impact on the reentrancy of the concurrency strategy, so these two variabilities are orthogonal. The resources covered by a concurrency strategy are also independent of how the locks are offered, so resources are orthogonal to control flow and reentrancy. Transition of locks, whether they can abort or are in effect immediately is independent of the resources, control flow or reentrancy. Therefore, this criterion is also orthogonal to all others. Syntax changes when the resource granularity changes, or when the control flow changes, or sometimes even when the reentrancy changes. Therefore syntax is partly defined by all other axes. Syntax is not a pure orthogonal variability. Figure 5.11 illustrates this.

If we now need to take into account the different concurrency strategies we must understand that layering influences everything. Syntax changes to support an extra layer, resources change since extra resources are added, control flow changes because different layers interact with each other in different ways. Transition changes since lower layers need to be rolled back when an

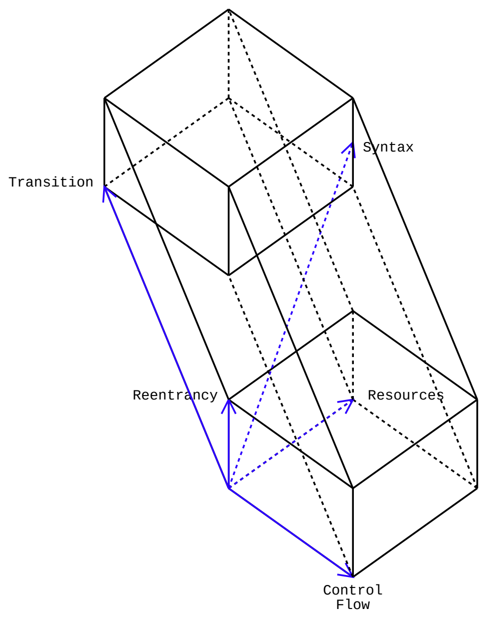


Figure 5.11: Projection of a hypercube illustrating the variabilities of a one layered concurrency guarding strategy.

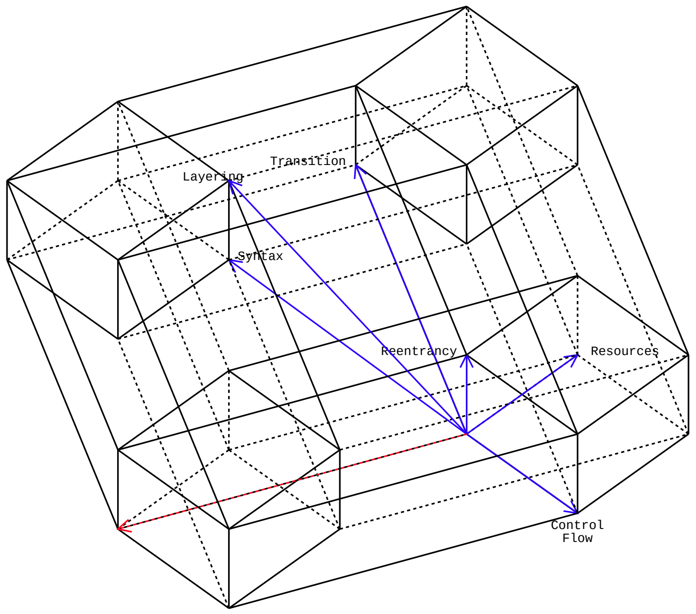


Figure 5.12: Projection of a hypercube of layered concurrency guarding strategies.

upper layer decides to roll back. Figure 5.12 illustrates this. The red arrow in the figure indicates an extra axis/dimension for layering. The axis we will work on on the other hand is slanted in the other 5 dimensions because it is impossible to change layering without interfering with other properties. On the other hand, we cannot express other properties solely by means of control flow, transition, reentrancy and resources, because an extra layer needs to reason about other resources.

However, this view on the different variabilities, should not be considered to be complete. It suffices for our goal, which is to select a set of concurrency conflicts. For instance, syntax should not necessarily define a separate variability, while a missing variability might be timing. We have limited ourselves to these because they capture a great number of possible concurrency strategies and because the resulting concurrency strategies can occur in practice.

5.12 Summary

IN THIS CHAPTER WE HAVE IDENTIFIED COMMONLY occurring concurrency problems based on a whiteboard on which multiple different actors can draw any figure they want. We will continue to use the whiteboard as a running example throughout the thesis.

When concurrency is, with any technique available, managed at a central place, and everybody adheres to the concurrency strategy, there cannot be much concurrency problems. The reason behind this is that the whole program uses the same concurrency strategy and this strategy is, when well designed, suitable for the program in question. Unfortunately, we saw that this requirement does not hold for open distributed systems.

The problem with concurrency is that it cannot be modularized. We cannot easily say where the concurrency management should be placed, nor can we offer one interface to the outside world which hides all our internal concurrency problems. When we place two deadlock-free programs in the same environment, it is possible to have a deadlock between both programs.

We investigated a number of problems with their solutions, and saw that these solutions in turn give rise to new problems:

- $\leadsto$ problem: race conditions $\rightarrow$ solution: atomic operations
  - solution: non-waiting atomic operations $\rightarrow$ problem: starvation
  - solution: waiting atomic operations $\rightarrow$ problem: deadlocks
- $\leadsto$ problem: performance $\rightarrow$ solution: locking per resource $\rightarrow$ problem: livelocks
- $\leadsto$ problem: partial failure $\rightarrow$ solution: transactions
- $\leadsto$ problem: multiple components $\rightarrow$ solution: distributed transactions.

We identified 6 parameters, which are sufficient to describe all the different approaches we have seen. They are: *syntax*, *control flow*, *reentrancy*, *resources*, *transition* and *layering*. In the next chapter we will use these variabilities to select a number of interface conflicts.

Chapter 6

Conflicting Interfaces

THIS DISSERTATION IS FOCUSED around interface conflicts. Interface conflicts arise at the moment two interfaces offer a similar, but not exactly the same, behavior. In such situations those interfaces will be able to contact each other, but the performed actions will probably induce incorrect behavior. We try to show that the concept of creating an automatic intelligent adaptator is possible. To show this we need a set of conflicts on which we can demonstrate the correct working of our technique. Doing this in a formal way by selecting a statistically significant subset of possible adaptors is impossible, therefore we will design a set of conflicts. To this end, we will create a set of concurrency strategies based on the variabilities presented in the previous chapter. Initially we will investigate conflicts between two non matching interfaces. Afterward we will investigate how multiple concurrency strategies can conflict. For every conflict presented we will investigate, as an exercise to explore the domain, how this conflict can be manually solved.

6.1 The Nature of Interface Conflicts

BEFORE WE DELVE into creation of an intelligent adaptor we need some idea of what such an adaptor might and might not do. We will therefore illustrate that it is not always possible to write an adaptor that will mediate differences between conflicting interfaces. It will also become clear that most adaptors depend largely on the usage involved and that some adaptors can be written between two conflicting interfaces but will fail to work when used in combination with multiple conflicting interfaces.

In the explanation below a *one-one* conflict refers to a conflict between two components. In our case, one being a server component and the other being a client component. A *one-multi* conflict refers to a conflict that arises between one server component and multiple client components. A *multi-multi* conflict refers to a conflict in which multiple servers and multiple clients are involved. We will start with a typical one-one conflict. A client component is a component that *requires* a the availability of a concurrency strategy. A server component is a component that *provides* a concurrency strategy.

6.1.1 Interface Usage is of Vital Importance

We will now explain that the way interfaces are used largely determines how an adaptor should be written.

A first trivial one-one interface conflict arises whenever a client expects a counting semaphore from the server, but where the server only offers a binary semaphore. In such a case there an interface conflict can arise. Consider the usage scenario depicted in figure 6.1: at the moment the client locks the server a second time, the server will still think it has been locked only once. The next unlock from the client will release the server, while the client still thinks it has a lock. Adapting such an interface conflict can be easily done. The code for such an adaptor is pictured

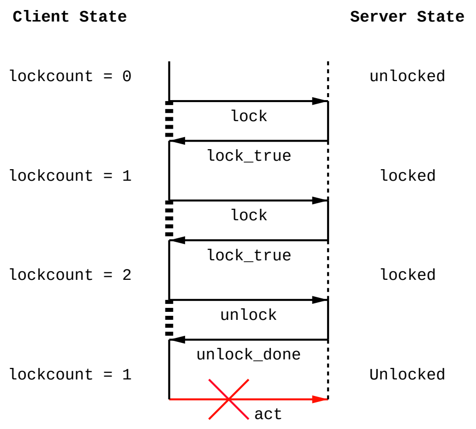


Figure 6.1: A simple conflict between a client that expects a counting semaphore on server side and a server that only offer a binary semaphore.

in algorithm 17: basically what is done is that the adaptor keeps a counter with which it determines in what state the client is supposed to be and in what state the server is supposed to be. When necessary the adaptor will contact the server to change its state to the clients expected state. Basically this means: the adaptor will only forward a lock request when the server is still unlocked and the client wants a lock. Also the adaptor will only forward an unlock request when the client has released its *last* lock.

Algorithm 17 An adaptor that mediates between a counting and a binary semaphore.

```
$counter := 0$
incoming $Lock()$
  **if** $counter = 0$ **then**
    $server..Lock()$
  **else**
    $LockTrue()$
incoming $LockTrue()$
  $counter := counter + 1$
  $client..LockTrue()$
incoming $Unlock()$
  **if** $counter = 1$ **then**
    $server..Unlock()$
  **else**
    $UnlockDone()$
incoming $UnlockDone()$
  $counter := counter - 1$
  $client..UnlockDone()$
```

At the moment this adaptor is only used between a waiting client and a waiting server there will be no problems. Problems arise when the client works asynchronously. In such a situation the client would send out a number of lock requests at the same time, each of which will be forwarded to the server. Which brings the adaptor in an uncertain state: a number of the lock requests will return `LockFalse` while others will return `LockTrue`. To solve this problem we should add a queue that can hold messages until they are ready to be processed. This effectively means that we need to process the first lock request completely before we can pass through other requests. So we set the appropriate requests on hold.

Again, within another usage context, the client interface and server interface may be connected immediately without the need for an adaptor. This happens when the client simply doesn't lock a certain resource more than once. These initial examples illustrate that the way interfaces are used largely determines how an adaptor should be written.

6.1.2 Not All Conflicts can be Mediated

The next example will show that it is not always possible to write an interface adaptor. As an example we go back to the the line actor and moving dot actors. The line actor expects a nested, non-waiting locking strategy from the server, with a granularity at *squares*. The server on the other hand provides a nested, waiting locking strategy for the *whole server*. With respect to the one-one conflict between both interfaces, an adaptor can easily be written. A lock of a square is translated to a server lock and released at the moment the adaptors lock count reaches zero.

However, in practice the whole server may be locked forever by the line actor. Such a situation can arise at the moment the line's trail is kept locked over a single movement. The following sequence illustrates how this can happen:

1. line has locked the positions (5,5..15) and the trail (4,6).
2. line locks position (6,5..15) and the trail (5,7). The position (5..6,5..15) and (4,6) are locked. Position (5,7) is locked twice.
3. the line releases its original position and its original trail. Now the positions (6,5..15) and (5,7) are locked.

In the above scenario the line actor will always have a lock somewhere on the whiteboard and will never release *all* its locks. If we are locking squares this doesn't matter; in this case however, the whole server will be locked. This shows that certain interfaces cannot be adapted to each other. Whether we can only mediate trivial differences or whether we can also mediate more difficult interfaces is an important question.

6.1.3 Adaptors Need to Cooperate

The third example illustrates how concurrency interface conflicts can be solved between two interfaces (one-one conflicts) but afterward fail to work in the global picture (one-multi and multi-multi conflicts), unless other interfaces are also adapted using a similar technique.

Consider therefore the case where the server supplies a locking granularity at *squares*, and a client that requires a locking at *whiteboard* level (this is the enter/leave locking strategy). Adapting the client to work with the server may seem easy: it suffices to lock all the squares on the whiteboard. The problem with this is that locking all fields is a relatively large task, especially when other actors are also on the whiteboard. We cannot expect to lock all squares without any other square being locked by somebody else. What we do in such a situation can differ, but in both cases it will provide a sub optimal solution:

- ⇝ Try and release again: If a lock cannot be acquired release all locks we already have and try again at a later time. By doing so, the client will almost never have a lock on the whole server when other actors are working on the server as well.
- ⇝ Be persistent and once locked never release: If a lock cannot be acquired we wait until the lock is released again. It is clear that this can lead to a deadlock.

So in any case, this simple solution: lock all squares, does not work. Luckily this can be remedied by adapting all connections to the server. If all connections are adapted to obtain a field lock instead of a square lock, we can easily make this situation to work. (provided that there are no line actors that wants to lock the whiteboard whole the time of course)

6.1.4 Brief Summary

In this section we described some of the intrinsic properties of interface adaptors:

- ⇝ Adapting interfaces to each other depends largely on how the interface is used.
- ⇝ We cannot always adapt any interface to any other interface: some at first sight simple adaptors will not work in practice.
- ⇝ When adapting two interfaces, indirectly linked other actors may need to be adapted also.

We will initially not investigate the latter since creating two interfaces that go along with each other is already quite a difficult thing to do. In chapter **10** we will investigate how multiple different adaptors can be made to work together.

6.2 A Selection of Interface Conflicts

AN INTERFACE CONFLICT is a conflict between two (or more) interfaces. This means that the space of interface conflicts is a space with two axes for one-one conflicts and more axes for one-multi and multi-multi conflicts. Every axis in conflict space represents all possible interfaces (which are either provided or required). We have illustrated in section 5.11 that every interface can be represented by a number of (not necessarily orthogonal) variabilities. This means that conflict space is at least 12 dimensional: 6 dimensions per interface, and at least 2 interfaces for every conflict. For one-multi and multi-multi this is even larger.

If we want to show that our approach to intelligent adaptors is valid and can provide a help within the domain of interface conflicts we should select a number of representative interface conflicts. The problem with this is that 'representative' is not well defined.

6.2.1 Selecting a Set of Interface Conflicts

In practice human programmers will have the tendency to implement a certain functionality according to existing functionalities or according to what they need. In fact there will be a lot of interface conflicts but not as many as covered by the entire conflict space. For example, a programmer will almost never implement a two layered concurrency strategy where the second layer has a granularity *lower* than the first layer. It is important to realize that adaptor generation algorithms are useful if they cover a lot of conflicts that will occur in real life. When an adaptor generation algorithm is unable to generate an adaptor for a conflict that almost never occurs, it still remains a useful algorithm.

The problem with selecting a set of real interface conflicts is that the behavior of programmers changes over time and that we don't have any information about current often recurring interface conflicts. So, we cannot select a number of existing often recurring interface conflicts.

6.2.2 Designing a Representative Subset of Conflicts

To design a representative set of conflicts we will fall back to the orthogonalities we have identified earlier. The 6 different orthogonalities (which are by no means exhaustive) allows us to investigate how we can mediate slight difference, that is differences on only one orthogonality, and create more complex conflicts by altering multiple variabilities simultaneously. Therefore, we will create a conflict-set which

- ⇝ contains conflicts within the given domain of *concurrency* only.
- ⇝ contains *basic* conflicts, that is conflicts on only one variability. We need these because we want to know how small conflict changes compare to solving the conflicts. Hopefully this will give some indication to the structure of the problem space because a number of variabilities are orthogonal.

- ⇝ contains *combined* conflicts, that is interface conflicts that conflict on more than one variability. This is an important issue, because we don't know anything about the conflict space, therefore we certainly need to randomly select a number of conflicts. How these are selected will be explained later.
- ⇝ contains conflicts that *can* be solved by humans as well as conflicts that *cannot* be solved, not even by humans. We will actively search for conflicts that cannot be solved, because this also gives some indication on the structure of the problem.

Afterward this conflict set will be expanded with other conflicts ...

- ⇝ that *can* be solved in *one-one* and can be readily extrapolated to one-multi and multi-multi situations.
- ⇝ that *can* be solved in *one-one* but *cannot* be solved, or are substantially much more difficult to solve, in one-multi or multi-multi situations.
- ⇝ that *cannot* be solved in *one-one* but *can* be solved within one-multi and multi-multi situations by means of cooperation.
- ⇝ that *cannot* be solved in *one-one* and *cannot* be solved in one-multi or multi-multi situations.

Depending on the observations with these conflict sets we can create a new conflict set that will provide us with more information, when necessary. The remainder of this chapter will focus on generating the set of 'designed' conflicts, such as conflicts on one variability, or conflicts that meet certain other requirements. The set of randomly selected interfaces will be presented in chapter 3.

6.3 One-One Conflicts on One Variability

AS DESCRIBED IN SECTION 5.11, there are 6 variabilities that we identified for concurrency strategy interfaces. We will now try to obtain possible as well as impossible to solve conflicts on every variability. An impossible conflict is a conflict that *cannot* be mediated by any possible adaptor. An example of this can be found in section 6.1.2. The conflicts that are not impossible to solve are possible to solve.

6.3.1 Syntax

Server	Client	Variability
Squares	Squares	Granularity
in Lock(<X, Y>) **out** LockFalse() / LockTrue()	**out** Lock(<Y, X>) **in** LockResult(<Res>)	Syntax
Non-waiting	Non-waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.1: One-one simple syntax conflict

Syntax conflicts can easily be found. We will use a syntax conflict between a non-waiting, non-nesting client and server as illustrated in table 6.1. The server requires the client to define a position on the whiteboard in (x, y) order, while the client thinks the server works in (y, x) order.

The server returns `LockFalse` when the lock could not be obtained; `LockTrue` when the lock could be obtained, from the server. As a result the client expects a `LockResult(<Boolean|Res>)`, where `res` contains either *true* or *false*. This interface conflict can easily be remedied. For every incoming or outgoing position pair the adaptor needs simply to swap the two coordinates. When an incoming `LockFalse` or `LockTrue` comes back a new message `LockResult` should be sent out.

Server	Client	Variability
Squares	Squares	Granularity
in Do(<Action:0>, <Pos: X + Y * 100>) in Do(<Action:1>, <Pos: X + Y * 100>) out Do(<Action:2>) out Do(<Action:3>) out Do(<Action:4>)	out Lock(<X, Y>) out UnLock(<X, Y>) in LockFalse() in LockTrue()/UnlockDone() in UnlockFailed()	Syntax
Non-waiting	Non-waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.2: Encoding conflict

The conflict presented in table 6.2 is a bit more difficult. The server uses some kind of calling channel over which messages should be encoded. For example, a lock request should be encoded to the call `Do(0,` $x + y * 100$`)`, an unlock request should be encoded to the call `Do(1,`$x + y * 100$`)`. Messages from the server to the client are also encoded through the same calling channel, with the difference that some different returns are mapped to the same return value. In this case: `Do(3)` is send out to notify success. In case of failure the server will send out an appropriate action. Mapping these two onto each other is difficult because we need the ability to encode the outgoing x and y coordinates to one position, this requires some mathematical functions. A second difficult thing for an adaptor is distinguishing a `SuccesLock` from a `SuccessUnlock` message. This requires some knowledge about the context, more specifically what the previous message was.

6.3.2 Reentrancy

A reentrancy conflict is a conflict that occurs due to the fact that somebody is requesting something from the server, while the server is already doing such an action. Recursion (with some form of stack management) is a typical case of reentrancy; shared global states often do not offer reentrant behavior.

Server	Client	Variability
Field	Field	Granularity
Waiting	Waiting	Control Flow
Non Nested	Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.3: Simple reentrancy conflict

The first reentrancy conflict presented in table 6.3 is the example we explained in the beginning of this chapter (see figure 6.1). A conflict arises at the moment the client expects the server to be a counting server, while in practice the server is a simple non counting semaphore. At the moment a client locks the server twice and then unlocks the server, the server will be unlocked while the client still thinks it has a lock upon the server. This is a conflict that often can be solved quit easily by means of keeping a counter within the adaptor.

Server	Client	Variability
Field	Field	Granularity
Non-waiting	Non-waiting	Control Flow
Non Nested	Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.4: Asynchronous reentrancy conflict

The second reentrancy (table 6.4) conflict is a bit harder to solve. Like the previous conflict, the server only supports a binary semaphore, while the client expects a counting semaphore. The big difference now is that the client actively sends out a lot of lock requests and afterward gathers the answers. So the client works asynchronously. As discussed in the beginning of this chapter writing an adaptor between both concurrency strategies is possible, but is a bit more difficult since we need the ability to set incoming messages on hold, until the server answers.

6.3.3 Control Flow

Control flow conflicts are conflicts that arise from the fact that some messages will wait before returning while others don't wait.

Server	Client	Variability
Squares	Squares	Granularity
Waiting	Non-waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.5: Simple control flow conflict

The first control flow conflict (table 6.5) concerns a server offering a waiting locking strategy while the client expects a non-waiting locking strategy. If the client doesn't behave too unconventional, this will work since the client will always get a `LockTrue` back from the server. It is in fact not a real *conflict* because the interaction between both will work out well, even without an adaptor.[1]

[1]Some confusion can arise here. We assume that a non waiting server always returns either LockTrue or LockFalse. A waiting server will only return when the lock could be obtained, hence return a LockTrue.

Server	Client	Variability
Squares	Squares	Granularity
Waiting	Non-waiting Expects to do things while idling	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.6: Control flow conflict when idling

Table 6.6 illustrates what can go wrong between a waiting server and a non-waiting client. It has to be said that the example is a bit far fetched but can actually occur in practice. The unconventional thing this client does, is relying on the request-lock to return *false* sometimes. It relies on this to advance with some other important virtual thread. If the server never returns such a `LockFalse` the second virtual thread will cease to work. This is in fact a scheduling conflict due to the control flow between the two components.

Server	Client	Variability
Squares	Squares	Granularity
Non-waiting	Waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.7: Control flow conflict that is almost impossible to solve

Another interesting conflict, presented in table 6.7, is where the server offers non-waiting locking semantics and the client expects waiting locking semantics. The only thing an adaptor can do in this situation is start polling the server until the server returns `LockTrue`. Since this generates lots of network traffic this is clearly not a good solution. It brings down the network and the adaptor possible gets locked out when at too large a distance from the server. Probably there is no good general solution to this problem. Luckily solutions can be found within certain environments. Suppose we are working on a local network where packets are queued in a fair way, the server may be able to do exactly what is required without any explicit provisions for it.

6.3.4 Granularity

Granularity conflicts arise at the moment the *size* of resources differs. Sometimes a resource encompasses a whole server, while in other cases a resource description refers to a number of underlying data structures, while again in other cases a resource is directly mapped onto the underlying data structures. For our whiteboard example this means that a resource can be either a single square, a line or the whole whiteboardf.

The granularity conflict presented in table 6.8 concerns a server that offers a locking granularity at the level of lines (vertical ones). The client requires a locking granularity at the level of squares. Since the granularity that the client requires is smaller than the granularity offered by the server it is not too hard to interface the client with the server. Instead of locking a certain (x, y) position, the client can lock a certain line x. The client is sure that the position to lock is certainly locked.

Server	Client	Variability
Lines	Squares	Granularity
Non-waiting	Non-waiting	Control Flow
Nested	Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.8: A simple granularity conflict

Server	Client	Variability
Lines	Squares	Granularity
Non-waiting	Non-waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.9: A more difficult granularity conflict

The granularity conflict presented in table 6.9 is of a more difficult sort to solve. Here the server offers a non nested locking strategy for lines, while the client requires a non locking strategy for squares. This is the same as the previous granularity conflict, the only difference between both lies in the reentrancy. The fact that the server is non nested complicates a possible adaptor a lot. To illustrate this, think about locking two points at the same vertical line. In such a case, the first lock request will be translated to a lock on the given line, while the second request will be translated also to a lock on the same line. Since both are the same line on the server. Unlocking the first point will also unlock the second point immediately. To solve this the adaptor should keep track of positions it thinks are locked and map this on the server. The adaptor needs some form of memory.

Server	Client	Variability
Squares	Field	Granularity
Non-waiting	Non-waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Immediately	Transition

Table 6.10: An impossible to solve granularity conflict

The conflict presented in table 6.10 is impossible to solve. The server offers a granularity at the level of squares, while the client expects a granularity at the level of the full whiteboard. Although a solution might seem easy, it won't work properly. One might think to write an adaptor that simple starts locking all squares within the whiteboard. Only when all squares are locked is the field considered locked and a `LockTrue` is sent to the client. The problem with this solution is that there will be other actors on the whiteboard (otherwise we wouldn't need to lock), so locking all resources can be a difficult operation.

Surprising about this example is that an adaptor *can* be generated between two components, while in practice when multiple components are running the adaptor will simply not work. The only good solution to this problem is either to add an extra layer or to agree between a number of components to lock everything in a certain order.

6.3.5 Layering

When guarding access to different components, concurrency problems often occur within the concurrency layer itself. To solve these kind of problems, extra layers can be added until we have reached a suitable solution. In this section we will only cover one extra layer, because other extra layers require other topologies, which do not fit within the one-one scenarios. We will discuss these in section 6.4.

When multiple layers are present, the conflict tables will contain a description of granularity, control flow, reentrancy and transition for each layer. The server in table 6.11, contains two layers. The first layer works at a granularity of squares, while the second layer (the red fields in the server column) works at the whole whiteboard. When presenting two (conflicting) interfaces we will put layers with the same granularity at the same level.

Server	Client	Variability
Squares	Squares	Granularity
Waiting	Waiting	Control Flow
Nested	Nested	Reentrancy
Immediately	Immediately	Transition
Field		Granularity
Waiting		Control Flow
Nested		Reentrancy
Immediately		Transition

Table 6.11: Simple layering conflict

The first conflict (table 6.11) occurs when the server request from the client to announce a series of locking operations and afterward finish the lock request operation. A series of square locking operations is announced by a single lock upon the whole server/whiteboard. Finishing the square locking operations is done by unlocking the server/whiteboard. In this conflict, the client in this case simply doesn't know that such a thing should be done. This conflict can easily be solved simply by wrapping every lock request within the required start and stop operations.

Server	Client	Variability
Squares	Squares	Granularity
Waiting	Waiting	Control Flow
Nested	Nested	Reentrancy
Immediately	Immediately	Transition
	Field	Granularity
	Waiting	Control Flow
	Nested	Reentrancy
	Immediately	Transition

Table 6.12: Difficult layering conflict

Table 6.12 covers a layering conflict where the server offers a single layer to lock resources. The client on the other hand expects from the server to offer some kind of server lock to start and stop requesting locks. This conflict cannot be easily solved by ignoring the start and stop operations. If we do so, we might end up with a set of deadlocks. The reason why one may need such an extra layer is to avoid deadlocks on the server. This implies that an adaptor should be able to detect deadlocks in some way and resolve them. Detecting them can be done with some form of timeout. Solving them will be a bit more difficult, therefore we will need to release all the locks we already requested, including the one pending at the moment. In this way, other waiting parties will be able to obtain their locks.

6.3.6 Transition

Transition refers to how locks influence resources. If changes made to a resource are in effect immediately, there is not much to say about transition: we say that the transition is immediate, because there is no time difference between *changing* a state and *being* in that state. On the other hand if changing a resource only results in actual changes when the lock is committed or rolled back there is a time difference. So in this case the transition is a *commit/rollback* transition.

Server	Client	Variability
Squares	Squares	Granularity
Non-waiting	Non-waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediately	Abort/Commit	Transition

Table 6.13: Simple transition conflict

The first illustration of such a transition conflict is illustrated in table 6.13. Here the server has no special transition provisions. It simply does immediately what is asked. The client on the other hand wants the ability to lock resources and then choose to *abort* or *commit* the lock. In both cases the lock is released. The resource state becomes either what is requested (a commit) or what the state of the resource was before the lock (an abort). Adapting these two different interfaces is no problem. This can be done by an adaptor that obtains the state of the resource immediately after locking.

Server	Client	Variability
Squares	Squares	Granularity
Non-waiting	Non-waiting	Control Flow
Nested	Nested	Reentrancy
Immediately	Abort/Commit	Transition

Table 6.14: Nested transition conflict

In the conflict represented by table 6.14, we have a nested locking strategy in combination with a commit/abort transition. This means that for every new lock on the same resource we can undo the changes. Suppose we lock a resource, with value A, and change that value to B; and afterward we lock that resource again and change that value to C, then this resource will be brought back to value B after the first abort and back to value A after the second abort. The conflict between two interfaces will pop up at the moment the client aborts a lock, while the serve is unable to undo the changes already made. This will leave the server in an unanticipated, possibly wrong, state. In essence mediating this interface conflict is similar to conflict 6.13, with the big difference now that the adaptor needs to have a stacked memory for every square on the board.

Server	Client	Variability
Squares	Squares	Granularity
Non-waiting	Non-waiting	Control Flow
Nested	Nested	Reentrancy
Immidiatelly	Immediately	Transition
Field	Field	Granularity
Waiting	Waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Commit/Abort	Immediately	Transition

Table 6.15: Typical transaction transition conflict

Scenario 6.15 covers a conflict where the server provides and the client expects two layers. The lowest layer, the level of resources, is a simple lock/unlock interface where it always seems as if the changes are in effect immediate. The layer above this resource-lock layer is the transaction layer where we can start a transaction, which we can either abort or commit. Committing means that all changes made to locked fields will be realized. Abort means that a rollback of all changes will be done.

The "conflict" here exists in the fact that the server offers a transaction interface, but the client simply doesn't want to use it. Adapting these is very easy: when the client requests an unlock operation the server is simply requested to commit the lock. The other way around on the other hand might be more difficult.

Server	Client	Variability
Squares	Squares	Granularity
Non-waiting	Non-waiting	Control Flow
Nested	Nested	Reentrancy
Immidiatelly	Immidiatelly	Transition
Field	Field	Granularity
Waiting	Waiting	Control Flow
Non Nested	Non Nested	Reentrancy
Immediatelly	Commit/Abort	Transition

Table 6.16: Another typical transaction trasnition conflict

In the conflict presented in table 6.16, the server offers solely a server lock that can be requested, but all changes are immediately effective. The client on the other hands want the possibility to roll back and commit. This seems to be a very difficult problem to solve because the adaptor needs to extract some knowledge from the server (what is the state of a resource) and needs to hold back changes to this state until a commit happens. Fortunately it isn't. The same technique as used with conflict 6.13 is appropriate here, as is a memory of all the locks obtained yet and released already.

6.4 One-Multi Conflicts

UNTIL NOW WE HAVE STAYED CLOSE to conflicts between two partners: a server that provides a certain concurrency interface and a client that expects a certain concurrency interface. In practice, there will be a server with a number of clients.

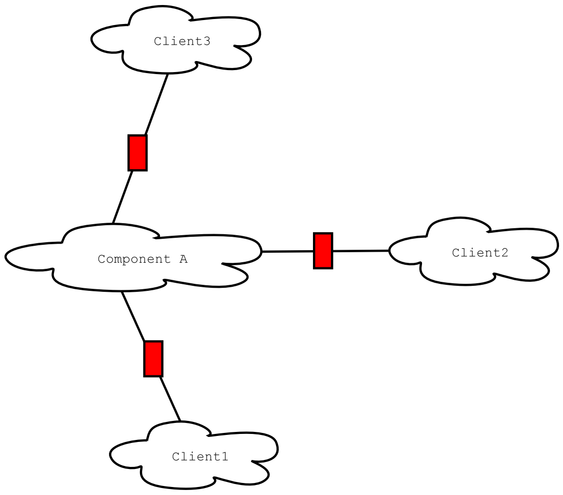


Figure 6.2: Non cooperating adaptors on all connections.

The problem we are now faced with is that every client connected to the same server might expect the server to offer another concurrency strategy. One might think that simply placing adaptors at all client-server connections would solve the problem (see figure 6.2). It is interesting to see that this is not necessarily the case. To illustrate this we have selected a number of interesting conflicts, for which a solution requires the *cooperation* between the different communication partners.

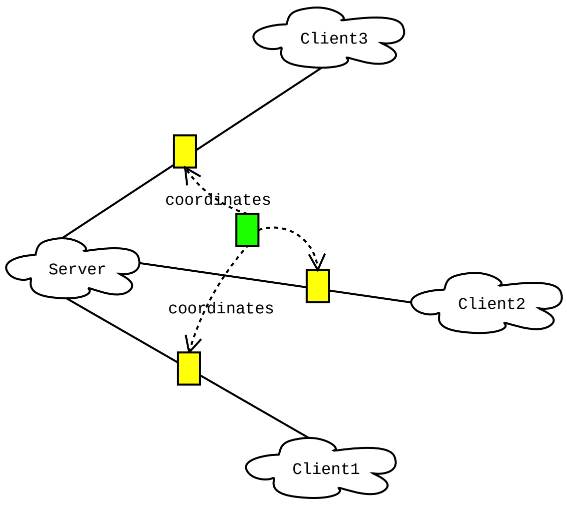


Figure 6.3: A set of adaptors on all connections that cooperate with each other.

To implement such a cooperating set of adaptors we resort to a more systematic view. The perfect solution would be as illustrated in figure 6.3, where all connections have an adaptor-drone whom behaviors are coordinated by one central adaptor-coordinator. The problem with this solution is that we not only need to create one adaptor, but a set of adaptor-filters, that all communicate with each other to cooperate and coordinate their global behavior. This requirement makes a solution substantially more difficult to implement. Therefore we will only generate a global adaptor that monitors all connections and coordinates the global behavior. This global adaptor will exhibit exactly the same behavior as is required from separate communicating adaptors. In practice one might be able to split up this one adaptor into smaller pieces and migrate those to better locations.

6.4.1 The Empty Server 1-x Conflict

Server	Client1	Client2	Variability
	Squares	Squares	Granularity
	Waiting	Waiting	Control Flow
	Nested	Nested	Reentrancy
	Immediately	Immediately	Transition

Table 6.17: The empty server one-multi conflict.

Imagine a server that does not offer any concurrency strategy, the most simple kind of server. The client on the other hand expects a certain locking strategy (see table 6.17). Interfacing these two conflicting concurrency strategies is easy in a one-one situation. The adaptor needs only to simulate all incoming synchronization calls in such a way that the client will continue working. E.g.: always return `LockTrue`. If we would place this dummy-adaptor at all the connections going from different clients to the same whiteboard, we will end up with concurrency problems because there is simply no overall concurrency strategy. To solve this problem all different clients will need to develop a certain concurrency strategy and adhere to that specification.

6.4.2 An Up-scale 1-x Granularity Conflict

Server	Client1	Client2	Variability
Field	Squares	Squares	Granularity
Waiting	Waiting	Waiting	Control Flow
Nested	Nested	Nested	Reentrancy
Immediately	Immediately	Immediately	Transition

Table 6.18: A granularity one-multi conflict.

A second illustration of these not-necessarily working generalized one-one adaptors can be found when the server offers a locking strategy with respect to the whole whiteboard and the client requires a locking strategy at the level of squares (see table 6.18). In this case an adaptor will simply translate every square-lock to a field-lock. The problem that can occur now is that in certain situations a client will always have a lock somewhere on the whiteboard. In this case, the adaptor will keep the whole server locked and no other client can do anything with the whiteboard. This is an illustration of a conflict that is hard to solve between different adaptors. It either requires substantial changes to the behavior of one client, or we need to override the full server locking semantics with a simulated more granular concurrency strategy.

6.4.3 A Down-scale 1-x Granularity Conflict

Server	Client1	Client2	Variability
Squares	Field	Field	Granularity
Waiting	Waiting	Waiting	Control Flow
Nested	Nested	Nested	Reentrancy
Immediately	Immediately	Immediately	Transition

Table 6.19: A granularity one-multi conflict.

A third problematic conflict occurs when the client wants to lock the whole whiteboard, but the server only supports locking of squares (see table 6.19). In such a situation, an adaptor will try to lock all the squares on the board, which will take a lot of messages but after all will work in a one-one situation. In a one-multi situation, this adaptor behavior could result in a livelock because two similar adaptors might want to lock the whole server in a non ordered way.

Solving this conflict requires the cooperation between the different adaptors. When a server lock should be obtained all adaptors could agree to lock only one reserved position on the board, which *means* that the whole server is locked. In this case all adaptors also agree to follow this convention. Another possibility would be agree to a certain order when starting to lock the whole whiteboard.

6.4.4 A Non Waiting Server 1-x Conflict

Server	Client1	Client2	Variability
Squares	Squares	Squares	Granularity
Non-waiting	Waiting	Waiting	Control Flow
Nested	Nested	Nested	Reentrancy
Immediately	Immediately	Immediately	Transition

Table 6.20: A non-waiting server one-multi conflict.

A fourth problematic conflict arises when a client expects a waiting locking strategy, but the server offers only a non-waiting locking strategy (see table 6.20). This requires the adaptor to actively poll the server constantly until the required lock is obtained. In a one-one situation this adaptor will always immediately get a lock, in a one-multi situation on the other hand this will result in a lot of unwanted network traffic and a non-fair scheduling behavior.

Therefore not only the interface conflicts between the client and the server should be solved, but also differences between the implicit interaction between the different clients.

6.5 Multi-Multi Conflicts

WITH MULTI-MULTI CONFLICTS we mainly aim at the situation where a number of components need a number of other components to reach a certain specific global behavior. We will again use the whiteboard example to illustrate this.

Suppose we have two board components and six actor components: a yellow moving dot, an orange moving dot, a red moving dot, a green moving line, a blue floodfill actor and a turquoise floodfill actor. The red moving dot together with the blue floodfill actor only move on the first board. The orange moving dot and turquoise floodfill actor work on the second board only. The last two actors: the yellow moving dot and the green moving line both work on both boards. (See figure 6.4).

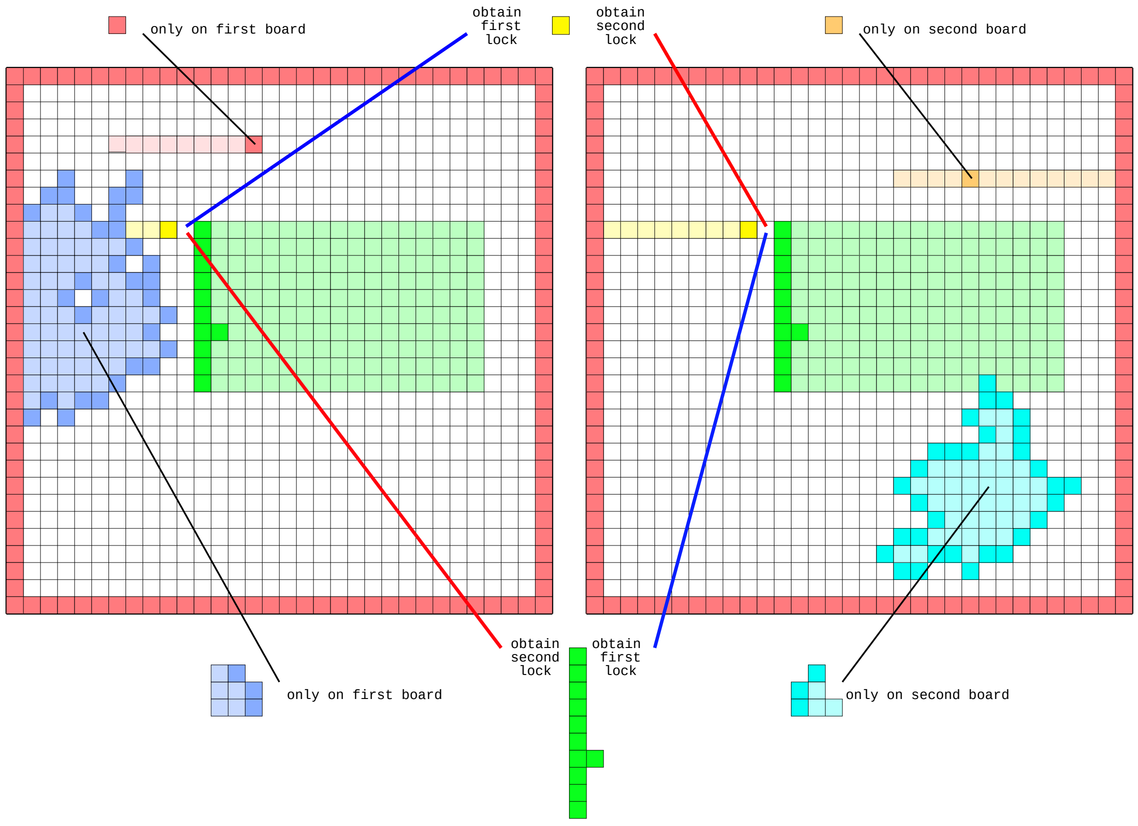


Figure 6.4: Peer 2 peer concurrency problem

The actors that only work on one board follow the logic explained earlier. The actors that work on both boards move by checking whether the next target position is free on both boards. Only when this is the case can the actor continue with its movement. The locking logic to implement this is done very simply by locking the first board and afterward locking the second board. It is clear that in this situation deadlocks can arise. Let's assume that the yellow dot actor first needs to obtain a lock on the first board and afterward on the second board, while the moving line actor first tries to obtain a lock on the second board. In such a situation a deadlock can arise.

This is a problematic situation because:

- ⇝ None of both servers can solve this problem, because none has the necessary knowledge. No server knows what an actor does with other servers.
- ⇝ None of the actors on both boards can solve this problem because they do not know each other and certainly not in which order they should obtain the locks.

1st Server	Red Client	Yellow Client	Green Client	Blue Client	2nd Server	Variability
Squares	Squares	Squares	Squares	Squares	Squares	Granularity
Non-Wait	Non-Wait	Non-Wait	Non-Wait	Non-Wait	Non-Wait	Control Flow
Nested	Nested	Nested	Nested	Nested	Nested	Reentrancy
Imm.	Imm.	Imm.	Imm.	Imm.	Imm.	Transition
Field	Field	Field	Field	Field	Field	Granularity
Waiting	Waiting	Waiting	Waiting	Waiting	Waiting	Control Flow
No-Nest	No-Nest	No-Nest	No-Nest	No-Nest	No-Nest	Reentrancy
Imm.	Imm.	Imm.	Imm.	Imm.	Imm.	Transition

Table 6.21: A multi multi conflict that requires the development of a distributed transaction adaptor.

The conflict presented in table 6.21 illustrates this. Here all actors adhere to exactly the same interface, nevertheless there will occur deadlocks if we create such a situation. The reason has already been explained above. The solution requires the development of a distributed transaction server, within the adaptor. By doing so, we introduce a second concurrency layer, which will guard the access to a number of different components. Of course it is perfectly possible that such a layer is already present and that we need to adapt the differences between those layers.

1st Server	Red Client	Yellow Client	Green Client	Blue Client	2nd Server	Variability
	Squares	Squares	Squares	Squares	Squares	Granularity
	Non-Wait	Non-Wait	Non-Wait	Non-Wait	Non-Wait	Control Flow
	Non-Nested	Non-Nested	Nested	Nested	Nested	Reentrancy
	Imm.	Imm.	Imm.	Imm.	Imm.	Transition
Field			Field		Field	Granularity
Waiting			Waiting		Waiting	Control Flow
Nested			No-Nest		No-Nest	Reentrancy
Imm.			Imm.		Commit/Rollback	Transition
			Components			Granularity
			Waiting			Control Flow
			No-Nest			Reentrancy
			Imm.			Transition

Table 6.22: Another real world multi-multi interface conflict.

The conflict presented in table 6.22 is another multi-multi conflict, containing two servers. The first server offers a lock on the whole whiteboard and places clients in wait until it is their turn. All changes made are in effect immediately. The second server offers a commit rollback interface. This is done by offering a transaction interface through which the client can obtain a transaction id: when this is done the client can lock any square it wants. When all changes are made a commit can be issued, otherwise an abort. The clients are implemented in the same 'realistic' fashion. The yellow and red moving dot actors simply require a non-waiting locking strategy without nesting. This is logical since the moving dot actor doesn't need to nest locks, and a possible livelock is easily remedied by choosing a random delay. The green moving line is also realistic in its approach. It needs to access two servers, so it wants a locking manager to be present. When done it requires the possibility to start a transaction on one or more servers, and when this transaction is obtained it can finally move the line. The last actor, the blue floodfill actor, requires a nested locking strategy that doesn't wait. Changes are in effect immediately. We want to use this large multi multi conflict to verify whether the algorithms we will develop will be able to solve this real life problem.

6.6 Summary

IN THIS CHAPTER we have presented the properties of interface conflicts. We have explained that every interface conflict largely depends on the context in which the interfaces are used. We have also explained that not all conflicts *can* be solved. And finally we have discussed the *necessity* of cooperating adaptors. We not only need adaptors between two conflicting interfaces, in a lot of cases we need adaptors that regulate and communicate their behavior, otherwise creating a good concurrency strategy may be impossible.

Afterward we presented many one-one interface conflicts, with a description of the feasibility of writing an adaptor. The presentation of one-multi interface conflicts focuses especially on adaptors that cannot simply be generalized from a one-one conflict adaptor. And finally, the multi-multi interface conflict discussion covers even more interface conflict examples.

Conflict	**Variability**	**Difficulty**		**Why**
		1-1	**1-x**	
6.1	Syntax	Eas	Eas	
6.2	Syntax	Med	Eas	Mapping between structural different communication channels
6.3	Reentrancy	Eas	Eas	
6.4	Reentrancy	Med	Eas	Requires busy-with flag and Q management
6.5	Control Flow	Eas	Eas	
6.6	Control Flow	Eas	Eas	
6.7	Control Flow	Eas	Imp	Unknown scheduling behavior
6.8	Granularity	Eas	Eas	
6.9	Granularity	Med	Eas	Needs to develop a memory
6.10	Granularity	Eas	Coop	Solution for 1-1 wont work on 1-x
6.11	Layering	Eas	Eas	
6.12	Layering	Med	Coop	Med: Needs to detect timeouts Coop: Can be better solved by cooperation
6.13	Transition	Eas	Eas	
6.14	Transition	Med	Med	Adaptor needs stack/square
6.15	Transition	Eas	Eas	
6.16	Transition	Med	Med	Adaptor needs stack/square and memory of acquired locks
6.17	Empty Server		Coop	Needs to implement a suitable locking strategy
6.18	Nasty client		Coop	Needs to neglect whole-time client lock
6.19	Livelock		Coop	Needs to cooperate to offer a server lock instead of square locks
6.20	Polling		Coop	Needs to develop token passing
6.21	Distributed Transactions		Med Coop	Adaptor needs to develop a distributed transaction server
6.22	Distributed Transactions		Hard Coop	Needs to develop distributed transactions and needs to mediate interface conflicts

Table 6.24: Overview of conflicts

In table 6.24 we present an overview of all the conflicts we have discussed. The first column refers to the summary-table of the conflict. The second column specifies which variabilities the conflict explores. The third and fourth column represent the difficulty to develop an adaptor to solve the conflict. The third column represents the difficulty to write an adaptor between 1 client and 1 server. The fourth column shows how difficult it is to get the one-one solution to work in cooperation with other (unknown) actors. Both columns use a number of abbreviations:

- ⇝ *Easy*; The adaptor can be written in a trivial way.
- ⇝ *Med*: medium; The adaptor can be written but is not trivial.
- ⇝ *Hard*: it is hard to develop a correctly working adaptor. The reason why is explained in the last column.
- ⇝ *Coop*: In the one-multi column, coop means that an adaptor can be written only if all participating components develop the same strategy and adhere to this strategy.
- ⇝ *Imp*: Impossible. It is impossible to write an adaptor in this situation.

The table contains three parts. The first part are the one-one conflicts, the second part are the one-multi conflicts and the last part are the multi-multi conflicts.

In the next chapter we will discuss how we will describe interfaces and which interfaces we will use.

Part III

Solution

Chapter 7

Our Approach

UNTIL NOW WE HAVE EXPLAINED how Petri-nets can be used to describe the behavior of interfaces, we have described which case we will use and which conflicts we will investigate. We will now explain which solution we propose to solve the problem of conflicting concurrency strategies. In general, we will do this by inserting an adaptor between the different communicating components. This adaptor will mediate the differences between the different interfaces. To do so, the adaptor will make use of three different modules: a liveness module, an enforce-action module and a concurrency module. Every module will be responsible for a certain functionality, which will meet certain requirements.

In this chapter we will first explain which assumptions we make on the components, then we will explain which requirements we place on the adaptor and last we will describe which modules we propose.

7.1 Assumptions about Petri-Nets and Components

BELOW WE INTRODUCE A number of important assumptions that form the basis of our research.

The first assumption we make is that the concurrency behavior of all involved components is documented by means of a Petri-net description. These Petri-net descriptions follow the guidelines presented in section 3.8 and are used by the adaptor to keep track of the underlying component behavior. For every component involved, a marking resides within the adaptor. Whenever a message m comes in from a component, the associated Petri-net will execute the transition corresponding to m. As such, the adaptor always has a correct representation of the state of the underlying components.

A second assumption is that the concurrency adaptor, which is placed between conflicting components, needs to work at runtime. Working at runtime means that the adaptor will choose certain synchronization actions to modify the behavior of the synchronization overall. However, before the adaptor can verify whether this overall synchronization behavior is correct, the adaptor should be able to obtain some kind of feedback. However, because the adaptor itself has no explicit supervisor, it needs some point of reference that can be used to verify its own behavior. Such a point of reference should allow the adaptor to verify a number of requirements of the mediated concurrency behavior, hence our point of reference should strongly correlate with the concurrency behavior. Therefore we will assume that the core functionalities of the communicating components is compatible, while the concurrency behavior of the different components might be incompatible. In our case we will separate these as follows:

1. All messages that deal with the core functionality of the components are said to be part of the *logic* interface. E.g.: the `Act` or `SetPosition` message.

2. A concurrency strategy is an algorithm that helps synchronize the behavior of some core functionality of the underlying component. We will say that all messages that deal with this behavior are part of the *synchronization* interface. E.g.: messages such as `Lock` and `Unlock`.

Thirdly, we assume that, as good software writing practices dictate, there is a clear separation between different functionalities. This implies that we should be able to say for every concurrency strategy whether it is required or whether it is provided. If a component requires a concurrency strategy we say that the component is a *client* component, if it provides a concurrency strategy it is said to be a *server* component. A second implication of a clear separation of functionalities is that we should be able to say for every message present on an interface whether it belongs to the synchronization behavior or to the logic behavior. Therefore we now define the $logic$ and $synchro$ sets formally. When given two Petri-net interface descriptions $N_0 = (T_0, P_0, A_0, \ldots)$ and $N_1 = (T_1, P_1, A_1, \ldots)$. We define $logic$ as the set of all in-transitions or out-transitions that involve no synchronization. Because all logic actions are known to be compatible, we can say that $logic \subseteq T_0 \wedge logic \subseteq T_1$. All the other transitions of N_0 and N_1 are assumed to be synchronization operations. Hence $synchro_0 = T_0 \setminus logic$ and $synchro_1 = T_1 \setminus logic$. We also assume that $synchro_0 \cap synchro_1 = \Phi$. If this might be the case a renaming operation in one of both transition should be used.

Fourthly, we assume that no concurrency strategy of any of the components within the conflict locks out certain core functionality, because this would make it effectively impossible to mediate the conflict.

7.2 Requirements for the Adaptor

GIVEN THE ABOVE ASSUMPTIONS we will now define the requirements of our concurrency adaptor. The three requirements we present have been introduced after doing preliminary experiments (discussed in chapter 11). Initially we tried to create an adaptor between different components simply requiring that no-conflict should arise. This however, resulted often a) in adaptors that either behaved in such a way that no communication with the server component occurred (by feeding always a `lockFalse` back to the originator), or b) in adaptors that mediated the conflict in such a way that race conditions were allowed.

Therefore we have introduced three requirements: the no-conflict requirement, the no-races requirement and the liveness requirement. Below we will explain these three requirements in more detail. Together, if satisfied, they lead to an adaptor that will mediate concurrency conflicts in an appropriate way. However, the three requirements we will present are not exhaustive. For instance other requirements with respect to timing, or requirements with respect to dead-locks might need to be added in other domains.

7.2.1 The No-Conflict Requirement

If we assume that the adaptor contains Petri-nets for all participating components and that the marking of every Petri-net represents the current state of the underlying component, then we can intuitively declare a certain situation to be a conflict whenever one logic transition can be executed within one Petri-net but not in the other.

Formally, two interfaces N_0 and N_1 are, given two markings M_0 and M_1, in conflict when a logic transition exists that is enabled in only one of both interfaces. We will use $\not\approx$ to denote a conflict between two Petri-net markings:

$$(N_0, M_0) \not\approx (N_1, M_1) \Leftrightarrow \exists t \in logic, \mid M_0[t\rangle_{N_0} \wedge M_1[t\rangle_{N_1} \tag{7.1}$$

When $(N_0, M_0) \not\approx (N_1, M_1)$ at the moment a $logic$ message arrives then we can be sure that the adaptor is not working correctly. We will define this to be the no-conflict requirement

⇝ ***No-conflict requirement*:** No sequence of actions taken by the adaptor should bring the associated Petri-nets into conflict.

The above definition is a 'good-enough' definition but not a 'cover-all' definition. The definition states that the adaptor is not working correctly if the precondition doesn't hold, but it does not ensure that *every* incorrect working adaptor is detected by the precondition. In practice, this requirement is difficult to check because there should be *no* possible execution trace leading to a conflict. This means, that if we want to verify this requirement, we must be sure that we have investigated all possible traces. We will come back to this issue later on.

7.2.2 The No-Races Requirement

The no-conflict requirement immediately results from the fact that we want to solve a conflict between interfaces. Nevertheless, it does not guarantee that enabled actions do not interfere with each other. To guarantee this we need to make sure that actions that are in the same critical section (which we will formally explain in section 10.2) are executed atomically. Therefore we need another requirement:

⇝ ***No-race requirement***

1. An adaptor should avoid *race*-conditions on actions and data. Critical sections should be entirely executed or not.
2. An adaptor should allow some degree of *interleaving*. An adaptor that consecutively executes every connected client until it terminates, cannot be considered a good concurrency strategy.

The no-races requirement, if satisfied, guarantees that no unwanted behavior as a result from race conditions will occur. However, often other requirements such as no-deadlocks or fairness could be also in place. Here we assume that, whenever appropriate the no-races requirement can be extended to include these extra requirements.

The problem with the above requirement is that it is difficult to define a critical section. Neither within the Petri-nets, nor within the components, there is a uniform notion of 'a critical section'. Therefore we define a critical section as a set changes that cannot be interrupted without bringing the corresponding Petri-nets in conflict. In section 10.2 (page 164) we will elaborate further on this and explain more intuitively why this is a good definition.

7.2.3 The Liveness Requirement

With the above no-conflict and the no-races requirements, we do not avoid adaptors that do not work. Adaptors that simply avoid any synchronization operation by always feeding a `LockFalse` back to the requester will match both previous requirements. However, such an adaptor is clearly not doing what is expected. Therefore we introduce the concept of liveness, or how well a component can proceed with its core functionality. The problem with the notion of liveness is that it can be either defined formally using Petri-nets, or informally based on some reward from the underlying component. Below we will present two possible definitions of liveness.

⇝ ***Formal Liveness Requirement***: In any situation should the involved Petri-nets be alive. A Petri-net is alive if every possible transition can always be enabled again in the future.

The first definition defines formally liveness on the Petri-nets involved. If the Petri-net is alive we assume that the underlying component is alive as well. Liveness on a Petri-net is defined as the number of transitions that still have an option to be enabled in the future. If under a marking a transition exists that *can* no longer be enabled, the Petri-net is not alive. However, this formal definition is very strong because not even all Petri-nets, received from the underlying components, will be alive when they start. For instance, a Petri-net with an `Init` transition might, after executing this transition, not be alive anymore, because this `Init` transition will never be enabled. This indicates that the formal definition of liveness might be a bit too strong.

⤳ ***Liveness Requirement***: The adaptor should try to obtain as much rewards as possible, as defined by the underlying component.

The second definition introduces rewards. Here we assume that the underlying component knows exactly what it wants to obtain in the future and indicates this by sending rewards to the adaptor. A component developer could specify this kind of information as checkpoints within the source code or as favorite transitions within the offered Petri-net. In both cases, the liveness requirement specifies that as many as possible positive future branches need to be executed, or in other words need to obtain as much reward as possible from the underlying client component. How the rewards are defined will be discussed in detail in section 9.2 (page 146) and will later on be used as the rewards within a reinforcement learning algorithm (section 9). This definition of liveness is similar to the description given in [Lyn96].

7.3 Pure Approaches

THE PROBLEM OF CONCURRENCY STRATEGY conflicts, embodied in our case, the availability of formal interface descriptions and the functional requirements of a solution now allow us to investigate which techniques are suitable to solve the problem. In general two techniques are possible: formal deduction of an adaptor and learning. We will investigate for every requirement the applicability of each technique. A pure application of one of both techniques will be impossible, as we will explain below, therefore in section 7.4 we will explain how we create a hybrid, modularized adaptor.

7.3.1 Automatic Deduction of an Adaptor

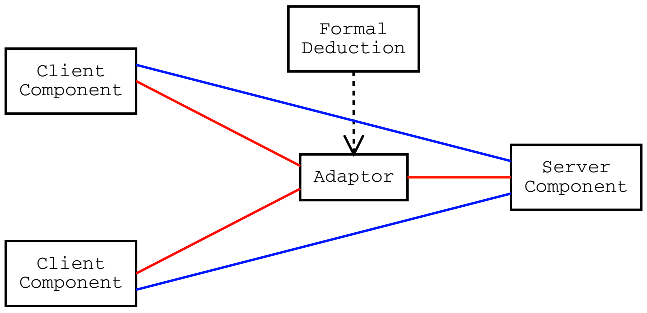


Figure 7.1: Formal deduction of an adaptor. Blue lines honor a logic protocol, red lines honor a synchronization protocol.

Given the formal Petri-net description of all involved components and assuming that we have a full formal definition of the requirements we might expect that it is possible to automatically deduce an adaptor, which satisfies all requirements and honors all restrictions. Such a formal technique would, given the Petri-nets as input, find some algorithm σ that could be placed in between the different Petri-nets such that all requirements hold. If we assume that σ itself is a Petri-net that directly links the two involved Petri-nets together, then we might not be able to verify the liveness requirement because the formal liveness property of Petri-nets is not yet known to be decidable or not (see section 3.10 on page 65). From this observation, it becomes clear that it is unlikely to create σ automatically.

7.3.2 Applicability of Pure Learning Algorithms

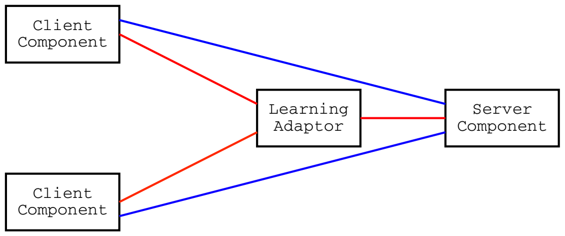


Figure 7.2: A fully learning adaptor. Blue lines honor a logic protocol, red lines honor a synchronization protocol.

In this section we will investigate whether a pure learning-algorithm approach (as pictured in figure 7.2**)** might be suitable. Learning algorithms are applicable in situations where formal techniques fail to offer solutions. A learning algorithm is typically a search algorithm that uses a number of heuristics (implicitly or explicitly represented) to find solutions. However, learning algorithms still remain probabilistic processes and proving that a probabilistic process will *only* result in adaptors that satisfy all requirements might be very difficult. Especially if the requirements state $\forall$-behavior (= $\nexists$-behavior), such as the no-races or no-conflicts requirements.

- ⇝ The no-races requirement is difficult to guarantee because a race should *never* occur.
- ⇝ The no-conflict requirement is also difficult to guarantee because a conflict should *never* occur.

These two no-requirements are, aside from being difficult to guarantee by a learning algorithm, also difficult to express numerically. However, in contrast to the other requirements, the liveness requirement can be measured numerically and spurious liveness failures are not a disaster.

7.4 Modularizing The Adaptor

IN SECTION 7.3 WE EXPLAINED that a purely formal or a purely learning based approach cannot solve the problem of concurrency adaptors. The main reason why a purely formal approach won't work is because of the liveness-requirement. The main reason why a purely learning approach won't work is because the no-conflict and no-races requirement cannot be guaranteed. Therefore, to solve the problem of creating a concurrency adaptor we will resort to an hybrid approach in which the no-conflict requirement and the no-races requirement are formally deduced, or validated, while the liveness requirement will be ensured by a learning algorithm. We will do this by placing these three requirements into separate modules, which together form the concurrency adaptor. Figure 7.3 illustrates how this could be done.

7.4.1 An Enforce-Action Module

The enforce-action module is placed between the server component and the remaining modules of the adaptor (see figure 7.3). Its main goal is to prepare the server for incoming *logic* messages. It does this by inserting appropriate synchronization messages in the message stream. For instance, when an `setPosition` arrives, and the server has not yet been locked, this module will lock it and send through the `setPosition` message.

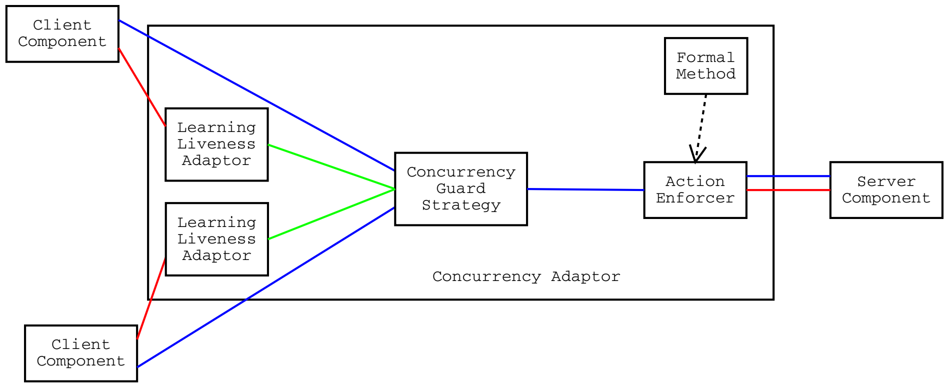


Figure 7.3: Cascade of components that avoids conflicts and honors a good working concurrency strategy. Blue lines honor a logic protocol, red lines honor a synchronization protocol. The green line is a separate protocol between the concurrency-adaptor and the learned adaptor.

This way, the adaptor fulfills the no conflict requirement, because it can always enforce requested *logic* actions upon the server component. If the client component expects a certain transition to be enabled (for instance `setPosition`), the adaptor should be able to enable this transition at the server component. Given the Petri-net description of an interface it is not too difficult to deduce formally what should be done to enable a certain action, or to reach a certain state. In chapter 8 we will explain how we do this. The responsibilities of the enforce-action module are:

1. *Bypass* all concurrency behavior of the server component by inserting synchronization messages whenever appropriate.

2. Present the adaptor-side only a *logic* interface that has only the core functionality of the server component.

7.4.2 A Liveness Module

A second important part of the concurrency adaptor are the liveness-modules. For every client-component there will be one liveness module. Every module is responsible for learning at runtime how to keep a client component alive, thereby honoring the constraints offered by the Petri-net. How the rewards are defined and assigned at runtime will be described in detail in section 9.2. The working of the entire liveness module is described in chapter 9. The responsibilities of the liveness module are:

1. Pass through any *logic* actions.

2. React on all *synchronization* messages by feeding some *synchronization* message back to the client, such that no synchronization message needs to be passed through to the server component.

3. Keep the client component *alive* by accumulating as much reward as possible.

4. If multiple actions can be taken in response to an incoming synchronization message, the different possible actions and enough information to recognize critical sections, will be presented to the concurrency module, which will choose an appropriate action.

7.4.3 A Concurrency Module

The concurrency module is placed between the two other modules. However, the problem of a good concurrency strategy is even more prominent than before, because any possible action chosen by the client will always be able to execute at the server (because of the enforce-action module), resulting in data and action races. Solving this problem is the responsibility of the concurrency module.

The concurrency module will be discussed in chapter 10. It receives from the client all the *logic* actions that should be forwarded to the server, from the liveness modules it receives the necessary information to be able to recognize critical sections. The concurrency module will use this information and make the different connected client components work together in a way that avoids colliding critical sections. The responsibilities of the concurrency module are:

1. *Proxy* the server component in such a way that nobody else can access the server component.

2. Deduce which *resources*, *actions* and *critical* sections are present.

3. *Interleave* critical sections to avoid race conditions.

4. Be *fair* with respect to the *liveness* of all connected client components.

7.5 Argumentation of a Correct Construction

THE PRESENTED MODULES make our initial requirements much more accessible. Two of the 'no-'-requirements (no-conflicts, no-races) are satisfied by known-to-work solutions and the liveness-requirement is satisfied by means of a learning algorithm. Cascading these three modules as pictured in figure 7.3 will result in a concurrency adaptor. In this section we will argument that this construction results in an adaptor that honors all the previously stated requirements.

7.5.1 Satisfying the No-Conflict Requirement

The no-conflict requirement declares that at no point in time the possibility of an immediate conflict should exist, that is a transition that is enabled in one Petri-net but not in the other. Formulated in terms of messages, that there is no message that can be sent but not received. The enforce-action module guarantees that it will always be able to execute any incoming *logic* message on the server-component. The two other modules present, the concurrency module and the liveness module, literally pass through any *logic* message. So any, possible *logic* request from any client component will always be executed. Hence no message exists that can be posted but not received, thus satisfying the no-conflict requirement.

7.5.2 Satisfying the No-Races Requirement

The concurrency module its main responsibility is recognizing and interleaving critical sections. So, if we know that no component but the concurrency module can contact the server-component, we know that there are no races. To argument this, let us assume the inverse, that the whole setup allows for two colliding critical sections. The only place where this collision could happen is at the concurrency module, because this is the *only* place where both critical sections will be present at the same time. The liveness modules work separately, hence cannot result in an incorrect interweaving of messages. The action enforcer module comes after the concurrency module and simply does not interleave any message, every incoming *logic* message is simply passed through. Thus normally all critical sections should already be serialized. Therefore, if two colliding critical sections can occur, the fault will lie in the concurrency module. Thus, if the concurrency module works correctly, then the interconnection of the three modules will also be correct.

7.5.3 Satisfying the Liveness Requirement

The liveness requirement states that all components should stay alive. This will be guaranteed by the interaction of the concurrency module and the liveness module. Every liveness module is responsible for keeping one component alive and because the concurrency module is, among other things, responsible for being fair with respect to liveness, all components will be alive.

7.6 Summary

IN THIS CHAPTER we have stated the assumptions we make. Firstly we assume that the core functionality of the conflicting components is compatible, secondly we assume that all components offer a Petri-net description of their concurrency behavior and that the concurrency behavior does not hide any core functionality. Thirdly we assume that a clear separation of functionalities exists with respect to the role of clients and server interfaces, and with respect to the role of every message involved in an interface. Messages that take part in the core functionality of a component are called *logic* messages. Messages that take part in the synchronization behavior are called *synchronization* messages.

After presenting the assumptions made, we have stated the requirements of a concurrency adaptor. These are the *no-conflict* requirement, the *no-races* requirement and the *liveness* requirement. With these requirements in mind we explained that neither a pure formal deductive, nor a pure learning approach will work to find a solution that will satisfy all requirements. Therefore we modularized the adaptor into three modules. The first module is an enforce-action module, which will solve the no-conflict requirement. The second module is a concurrency module, that satisfies the no-races requirement. And the third module is a liveness module that satisfies the liveness requirement. Finally we have argued that, given the assumptions and the responsibilities of the different modules, the interconnection of these three modules will result in an adaptor that satisfies all three requirements. In the following chapters we will describe every module in detail.

Chapter 8

Module 1: The Enforce-Action Module

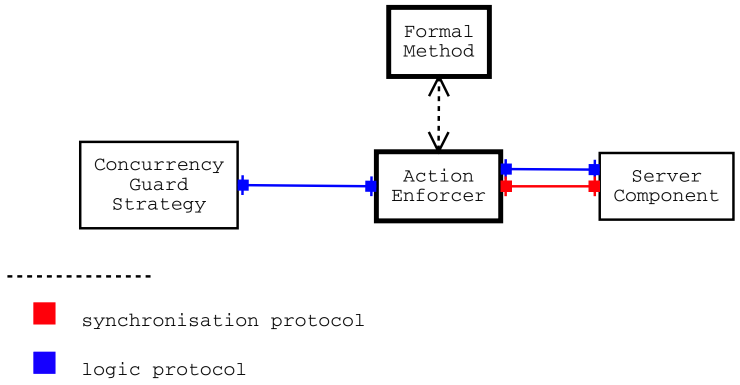


Figure 8.1: Bypassing a concurrency interface

IN CHAPTER 7 we have explained that we will construct an adaptor based on three requirements. Every requirement will be satisfied by one module. The enforce-action module will allow the adaptor to satisfy the no-conflict requirement. To this end, this module will bypass a provided concurrency interface by means of a logic deduction. Therefore a reachability analysis of the Petri-net description of the provided concurrency interface will be used. In this chapter, after presenting standard techniques to do such an analysis, we will explain how we will perform a reachability analysis by means of prolog.

8.1 Introduction

A REQUIREMENT FOR THE MODULE we will develop is that it should be able to receive any possible action from the *logic* interface and execute it on a server component. To do so the necessary *synchronization* messages should be generated automatically.

The adaptor itself has 3 ports (see picture 8.1). One port is connected to the concurrency module and provides/requires a *logic* interface. Another port is connected to the server-component and provides/requires a *logic* interface and a last port, providing/requiring a *synchronization* interface also connected to the server component.

In essence, there are two completely different techniques which we can use to shortcut the concurrency strategy. The first is the use of an on-line learning algorithm, which is suitable in this case because the learning algorithm can keep on trying to enable a certain action until it is successful. In the meantime the concurrency adaptor can be set to wait. The reward given here is straightforward and defined by the enabling of the required transition.

The second approach uses inductive reasoning to deduce what kind of actions should be taken in a certain context. This is the approach we will investigate in more detail because it has a faster response and works in most situations.

8.2 Formal Analysis

DEDUCING HOW WE CAN ENABLE a certain state within a petri-net, given its current marking seems typically a problem of reachability, however there are some differences. A reachability analysis of a Petri-net indicates whether we can reach a certain marking or not, often information is included how this marking can be reached. In our case we don't want to reach a specific marking, we only want to know how to enable a transition as a result of a certain marking. Formally, we want to find a way *how* to reach $M \in {}^{\bullet}t$ from M_0.

Nevertheless, in general formal reachability analysis not only gives a yes or no answer to the reachability question, but also gives a way to reach the target marking. Since we have a set of target markings (${}^{\bullet}t$), which we want to reach, we investigate the use of these formal techniques. Reachability is decidable [EN94]. However it might take a long time. Generally, it can be solved in exponential time. However for a lot of Petri-net classes different results exist. For instance:

- ⤳ If the petri-net is *symmetric* then the reachability problem is EXPTIME-complete (see glossary). A petri-net is symmetric when for every transition t there is a transition t' that undoes the effect of the first transition and returns to the original marking. This is a property which is often found in concurrency interfaces. Once a lock is obtained it is possible to release it again. However, aside from this intuition nothing guarantees that the petri-net also exhibits this behavior.
- ⤳ If the petri-net is *conflict-free* and *bounded* then reachability is decidable in P. (see glossary) A Petri-net is bounded when there is a maximum number of possible tokens present at a certain place (see section 3.10). A Petri-net is conflict-free if for every possible marking the net is persistent. A petri-net is persistent if for every place with more than one enabled output transition, the execution of one transition does not disable the other transition. In the case of concurrency interfaces this is highly likely because normally multiple transitions will not be enabled at once and the amount of resources remains fixed.
- ⤳ Reachability in timed petri-nets is NP-complete [LLPY97, BP96]).

Extensions to Petri-nets may complicate these results. For instance:

- ⤳ Our expression language is not a standard language so we don't exactly know whether all these properties hold with the given language. We assume they do, because the expression language used is a functional language which is not Turing complete and as we will experimentally observe most concurrency strategies can be written down as finite state machines.
- ⤳ Colored petri-nets make all these formal approaches a bit problematic. Expanding a colored Petri-net to a simple Petri-net may require an infinite explosion, so all these decidability criteria need to be investigated again.

There are a number of algorithms available to decide whether a marking M_1 is reachable from an initial marking M_0. Below we will briefly summarize them and explain why they do not fit our needs.

- ⤳ *Reachability graphs:* The reachability of marking M_1 from marking M_0 is often decided by creating a reachability-graph. Here the nodes of the graph contain a marking and the arcs contain the transition that brings one from marking a to marking b. Reachability is then decided by creating a matrix containing on the X/Y-axis all possible nodes/markings. The values within the matrix specify whether X is reachable from Y. With every step more positions are filled with 1, by taking the transitive closure of the reachability graph. This approach is formally very nice; a drawback however is that it takes too much time since often we only want to know whether A is reachable from B and not whether all possible A 's are reachable from all possible B 's.

- ⇝ *Unfolding Petri-nets*: With this approach a Petri-net is unfolded into another Petri-net, usually with an infinite but simpler structure. McMillan [ERV96], proposed an algorithm for the construction of a finite initial prefix of the Petri-net, which contains full reachability information. However, this information is difficult to generate, and can take a long time. Therefore we didn't investigate this track further.
- ⇝ *Exploiting symmetry*: It is possible to exploit symmetry between states by only looking at one side of the symmetry. This is especially useful for colored Petri-nets. Intuitively: within a dining philosopher Petri-net, there is a 4 way symmetry. There is no need in trying out every philosopher, which reduces the state explosion drastically. How this is done in practice is described in detail in [Jør].

Aside from all these techniques, there is a property of our problem which might also be exploited. In our case, we have the ability to search for a specific solution. As stated earlier, we want to know how to get from a certain marking M_0 to another certain marking M_1, *that enables transition t*. It is important that we are not even looking for the shortest path, but simply need to find one way to enable t.

Now, let us turn back our attention to Petri-nets. As explained in section 3.4.4, it is difficult to create a high performance Petri-net evaluator for colored Petri-nets because a transition is enabled if a suitable *combination* of input tokens exist. This is typically a search problem and we have argued that a logic engine (such as prolog) might be a suitable language to write a Petri-net evaluator in. Given the fact that most reachability analysis work for elementary Petri-nets, but have often difficulties understanding the possible expressions present in colored Petri-nets, it seems appropriate to use prolog as a logic engine to do a reachability analysis. This is what we will describe below.

8.3 Converting a Petri-Net Description to Prolog Rules

BEFORE WE CAN DEDUCE certain interesting properties from a Petri-net we need some representation of Petri-nets within prolog. We will now explain how we can convert Petri-nets to prolog rules. But before we do so, we will explain how we define our markings.

Markings are declared by a set of simple rules as shown in algorithm 18. A marking is represented as an association list of place-names and place-content. The content of a place is a list of tokens. The basic operations on markings are `del_marking(input_marking, to_delete, output_marking)` and `add_marking(input_marking, to_add, output_marking)`.
`del_marking` will remove a token from an input marking and create an output marking.
`add_marking` will add a token to a given marking. Markings can be either relative or absolute. A marking is *relative* if it only mentions the *necessary* tokens without including all possible other tokens that could also be present. A marking is *absolute* if all the tokens that are available are specified in the marking. The notion of a relative and absolute marking is necessary to be able to deduce which tokens should be present in a certain marking without actually having a real marking at hand. Without the notion of a relative marking the process of finding out how to reach a certain sub-marking might take a long time because the step predicates would try out all possible permutations within the offered absolute marking. A relative marking is well defined because Petri-nets do not allow an absence check of tokens, so we do not need to specify which tokens *cannot* be in a certain marking.

Algorithm 18 Definition of markings.

```
empty_marking([]).

del_marking([E|Y], E, Y).
del_marking([X|Y], E, [X|Z]) :- del_marking(Y, E, Z).

add_marking(X, E, R) :- append(X,[E],R).

dump_marking([]):-nl.
dump_marking([M|T]):-
  write(' '),write(M),nl,dump_marking(T).

marking_in([],_).
marking_in([Token|Rest],M1):-
    del_marking(M1,Token,M2),
    marking_in(Rest,M2).
```

Algorithm 19 How a `step` is defined for the `lock(X,Y)` transition.

```
step(lock(X,Y),M,N):-
  ( \+ var(M), \+ var(N),
    del_marking(M,[ready,[]],Markinga1),
    del_marking(N,[ready,[]],Markingb1),
    del_marking(Markingb1,[locking,[X,Y]],_));
  ( \+ var(M), var(N),
    del_marking(M,[ready,[]],Markinga1),
    add_marking(Markinga1,[ready,[]],Markinga2),
    add_marking(Markinga2,[locking,[X,Y]],N));
  (  var(M), \+ var(N),
    del_marking(N,[ready,[]],Markinga1),
    del_marking(Markinga1,[locking,[X,Y]],Markinga2),
    add_marking(Markinga2,[ready,[]],M));
  ( var(M), var(N),
    empty_marking(Markinga0), empty_marking(Markingb0),
    add_marking(Markinga0,[ready,[]],M),
    add_marking(Markingb0,[ready,[]],Markingb1),
    add_marking(Markingb1,[locking,[X,Y]],N)).
```

The convention we will use is that `step(T,M,N)` declares a valid transition T which brings marking M to marking N. E.g., `step(lock(X,Y),[],N)` will find all possible resulting markings `N` after executing the `lock(X,Y)` transition. This is illustrated in algorithm 19. Depending on whether M, N or T are known the behavior is different.

- ⇝ If both M and N are known, the predicate step(T,M,N) will check whether T is a valid transition between M and N. This is done by verifying the presence of all input tokens in M and the presence of all required output tokens in N.
- ⇝ If only M is known, N will match the resulting output. If multiple token-pulls are possible from M, multiple answers will be placed in N.
- ⇝ If only N is known, M will match the necessary input for T to result in N. If multiple possible inputs are possible they will all match.
- ⇝ If neither M nor N is known, The transition will create a *relative* marking M and N.

Together with definitions for steps and markings we need a way to categorize transitions, places, incoming transition, outgoing transitions, synchronization actions, logic actions and other. These categories are simply declared as facts. They are:

- ⇝ *transition:* declares whether something is a transition.
- ⇝ *place:* declares whether something is a place.
- ⇝ *action:* an action is a message that can be received or sent over a *logic* interface. These are return_joinactor(_), set_position(_,_,_), return_set_position, isfree(_,_), return_free_true and return_free_false.
- ⇝ *synchro:* a synchronization message. Every transition that is not an *action* is considered to be a synchronization transition.
- ⇝ *incoming:* a transition is an incoming transition if it is received from some external source. We have no choice but to accept incoming transitions and we cannot fire them ourselves.
- ⇝ *outgoing:* a transition is an outgoing transition if it is the result of another incoming transition. Incoming and outgoing declares the two directions in which a message can go.

An interesting property of these is that the parameterizations of a number of transition are mentioned as they are. For instance, the `lock` transition is declared as:

```
transition(lock(_,_)).
step(lock(X,Y),.....
```

where the two free variables X and Y can be filled in when appropriate. This fact allows us to reason about a colored Petri-net in an abstract way. This is a huge performance-improvement because, we can now easily check how we can enable `SetPosition(12,13)` without actually having a marking at hand. Therefore we need to check out how we can enable `SetPosition(X,Y)` and try to match the resulting *relative* marking, which will contain a token such as `locked(_,_)`. This allows for the creation of an abstract description how to enable the `SetPosition` token, something which would be very difficult with a completely expanded Petri-net.

8.4 Predicting the Future & Deducing the Past

WITH THE ABOVE PROLOG RULES in place we can relatively easy deduce what can happen given a certain marking. Essentially to know all possible future branches after one step, given a certain marking, we simple state:

Algorithm 20 Obtaining the possible future traces given an initial marking.

```
fwd_step(Marking,unknown([Transition,NewMarking])):-
    step(Transition,Marking,NewMarking),
    \+action(Transition).
fwd_steps([],[]).
fwd_steps([Head|Tail],[NewHead|NewTail]):-
    fwd_steps(Head,NewHead),
    fwd_steps(Tail,NewTail).
fwd_steps(expanded(Marking,Results),expanded(Marking,NewResults)):-
    fwd_steps(Results,NewResults).
fwd_steps(unknown(Cut),expanded(Cut,Results)):-
    Cut = [Transition,Marking],
    findall(Future,fwd_step(Marking,Future),Results).
fwd_steps(relative(M),Result):-
    fwd_steps([unknown([start,relative(M)])],Result).
```

```
:- step(Transition,marking,Result)
```

The result will give all possible answers, including the transition executed and the result after executing the transition. If we would like to know what possible branches exists after two steps we simply:

```
:- step(Transition1,marking,Intermediate),
step(Transition2,Intermediate, Result)
```

If we continue this line of thought we can easily see how we can enumerate all possible future traces given a certain depth and initial marking. Algorithm 20 shows how this can be done. The `fwd_step` rule will expand a marking into all possible futures. The `fwd_steps` takes a list of nodes, which can be either expanded or unknown. Unknown nodes are expanded one step further when executing `fwd_steps`. Expanded nodes are simply followed. With this we can construct a tree that can be expanded a bit further every time.

With this rule-set we can easily track down how we can enable a certain transition, or reach a certain marking. All we have to do is check whether the transition we want to enable (or the marking we want to reach) is present in one of the possible futures. If it isn't we can go one step deeper.

A small problem remains to be explained here. When finding out how we can go from M_0 to M_1 we can only investigate synchronization actions. For example, it should not be possible for such a trace to contain a `joinActor` message because the adaptor cannot choose to send out its message since it is part of the *logic* interface.

Algorithm 21 can be used to print out such a trace. For instance, given a certain initial marking, we can have a result such as:

```
start
.  lock(_G410,_G411)
.  .  lock(_G594,_G595)
.  .  lock_false
.  .  lock_true
.  .  return_unlock_false
.  return_unlock_false
.  .  lock(_G707,_G708)
.  .  return_unlock_false
```

This states very simply that, given the start situation, we can choose to execute `lock(_,_)` or `return_unlock_false`. It is clear that only the first one is under control of the client, because a

Algorithm 21 Printing out the trace tree.

```
trace_tree(_,[]).
trace_tree(Prefix,[Head|Tail]):-
    trace_tree(Prefix,Head),
    trace_tree(Prefix,Tail).
trace_tree(Prefix,expanded(Marking,Results)):-
    Marking = [Trans,_],
    write(Prefix),write(Trans),nl,
    string_concat(Prefix,' . ',NewPrefix),
    trace_tree(NewPrefix,Results).
trace_tree(Prefix,unknown(Marking)):-
    Marking = [Trans,_],
    write(Prefix),write(Trans),nl.
trace_tree(X):-
    trace_tree('',X).
```

Algorithm 22 Obtaining the possible past traces given an initial marking.

```
bwd_step(Marking,unknown([Transition,OldMarking])):-
    step(Transition,OldMarking,Marking),
    \+action(Transition).
bwd_steps([],[]).
bwd_steps([Head|Tail],[NewHead|NewTail]):-
    bwd_steps(Head,NewHead).
bwd_steps(expanded(Marking,Results),expanded(Marking,NewResults)):-
    bwd_steps(Results,NewResults),
    bwd_steps(Rest,NewRest).
bwd_steps(unknown(Cut),expanded(Cut,Results)):-
    Cut = [Transition,Marking],
    findall(Past,bwd_step(Marking,Past),Results).
bwd_steps(relative(M),Result):-
    bwd_steps([unknown([stop,relative(M)])],Result).
```

`return_unlock_false` must be issued by the server. The possible tracks from there on are either again to `lock` or to receive a `lock_false`, `lock_true` or `return_unlock_false`.

Given a Petri-net and a marking we can also, in a similar way, deduce which actions could have led to this marking. This is illustrated in algorithm 22 and is similar to algorithm 20.

8.5 Reachability Analysis: Forward and Backward Tracing

Algorithm 23 Checks if trace1 contains a marking that can be found in trace2. If there is a common marking, the way to reach it will be printed.

```
    trace_matches(Trace1,[Head|_]):-
        trace_matches(Trace1,Head).
    trace_matches(Trace1,[_|Tail]):-
        trace_matches(Trace1,Tail).
    trace_matches([Head|_],Trace2):-
        trace_matches(Head,Trace2).
    trace_matches([_|Tail],Trace2):-
        trace_matches(Tail,Trace2).
    trace_matches(expanded([Trans, _],Results),Trace2):-
        trace_matches(Results,Trace2),
        write('->'), write(Trans), nl.
    trace_matches(Trace1,expanded([Trans,_],Results)):-
        trace_matches(Trace1,Results),
        write('<-'), write(Trans), nl.
    trace_matches(unknown(Cut1),expanded(Cut2,_)):-
        Cut1 = [Trans1, Marking],
        Cut2 = [Trans2, Marking],
        write('->'), write(Trans1),nl,
        write('<-'), write(Trans2),nl.
    trace_matches(expanded(Cut,_),Rest):-
       trace_matches(unknown(Cut),Rest).
```

Algorithm 24 Finding out *how* to get from a given marking to another marking.

```
    solve_trace(ForwardTrace, BackwardTrace):-
        trace_matches(ForwardTrace,BackwardTrace).
    solve_trace(ForwardTrace, BackwardTrace):-
        fwd_steps(ForwardTrace,NewForward),
        bwd_steps(BackwardTrace,NewBackward),
        solve_trace(NewForward,NewBackward).
```

TYPICALLY, SEARCH ALGORITHMS, such as implemented in the forward or backward tracer, behave exponentially because the search tree expands exponentially. If the search algorithm looks at depth n, it will take approximately c^n time to find a solution, with c being a constant. So, if we can reduce the search depth by halve we can find a solution 2^n times faster. With the ability to trace into the future and into the past we can find a solution to the reachability problem faster. The only thing we need to do therefore is going forward (from M_0) and backward (from M_1) at the same time. When a common marking exists between both traces, then we have found a possible path. Algorithm 23 illustrates how we can check if two traces matches. Algorithm 24 will determine the different strategies possible to go from M_0 to M_1.

To illustrate the power of these rules, we will find out how we can enable position $(10, 20)$ on a whiteboard, given an initial marking. Figure 8.2 illustrates how the process works. On one track we have a forward reasoning (the top of the figure). This process determines that a

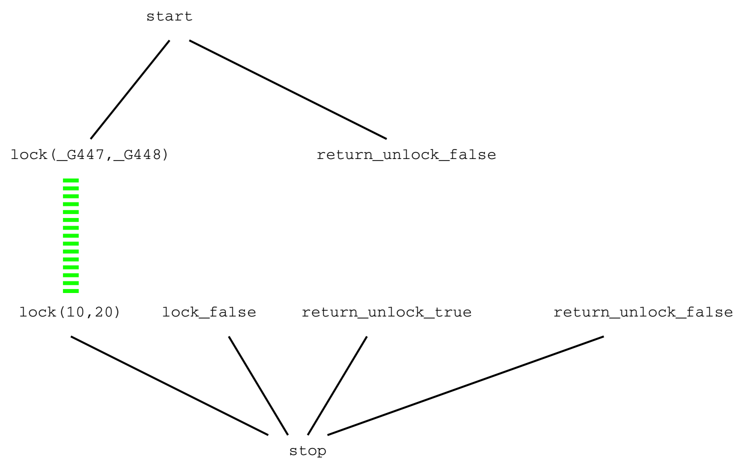


Figure 8.2: The trace-tree to reach a marking that bring position (10, 20) in locking.

possible future from the initial marking is `lock(_,_)`. A second future is the possibility that a `return_unlock_false` comes back. However, this event cannot be controlled by the client because it is an incoming event, thus we ignore this possibility.

The second process is a backward trace (the bottom half of the drawing), which correctly deduces that in the past a `lock(10,20)` could have been present, or a `return_unlock_false`, `return_unlock_true` or a `lock_false`.

The process of finding a matching trace stops here because the marking at `lock(_X,_Y)` can be unified with the marking at `lock(10,20)`.

8.6 Discussion

8.6.1 Implementation Notes & Performance

THE PRESENTED ALGORITHM has been tested with all the conflicts presented in chapter 6. However, because the prolog code was not integrated within Java, we had to test the code off-line. Therefore we obtained a start-marking from the Petri-net by exporting one from the Petri-net evaluator. After importing it into the prolog program, we asked the reachability program to enable a certain transition. In all cases the result was calculated immediately (no human observable time delay on a standard Intel processor). To a certain extent this is normal because most interfaces provide a certain functionality and are supposed to make state-changes easy and not difficult. An API which requires less messages to change a state than another API is clearly easier to use.

Now, the reachability analysis is calculated immediately, nevertheless it took some doings before we were able to come up with such a result. Prolog is a declarative language and is perfectly suited to find ways to prove statements. Every solution of such a prove is a way to reach a certain marking. Nevertheless, how the evaluator 'proves' a reachability statement can greatly affect the performance of finding solutions. This has forced us to insert our a) own delete operation, b) to make a distinction between delete and append, and c) split the step predicate into 4 parts depending on which variables are bound. We now explain the details.

Why not using 'delete/3' ?

The reason why we declare our own `del_marking` lies in the fact that swi_prolog (the implementation of prolog we have been using) is only able to delete one element from a list. The predicate `delete([1,2,3],X,Y).` results in only one answer:

```
X = 1
Y = [2, 3] ;
No
```

while the predicate `del_marking([1,2,3],X,Y).` results in all possible elements taken from the input list:

```
X = 1
Y = [2, 3];
X = 2
Y = [1, 3];
X = 3
Y = [1, 2];
No
```

By relying on our `del_marking` predicate we are sure that all tokens present at a certain place will be tried to satisfy the precondition of a transition. If we used the standard delete predicate this was not possible.

Why not using 'select/3' ?

Maybe it could be possible to use `select/3` as an unification of a `delete/3` and an `append/3` operation. This however would result in drastic performance penalties because there is a subtle difference between an `append/3` operation and a `select/3` operation. Select is defined as

```
select(X, [X | L], L).
select(X, [Y | L], [Y | R]) :- select(X, L, R).
```

We can indeed use this predicate to implement a delete operation. In fact `del_marking(X, Y, Z) :- select(Y, X, Z)`. However, if we would implement an `append/3` as the inverse of a delete, or by means of the select predicate then a simple append of two small lists, written down as `select(5, X, [6])` would result in:

```
X = [5, 6] ;
X = [6, 5] ;
No
```

This in contrast with the standard `append/3` operation, which will return only one answer: `append([5],[6], X)`

```
X = [5, 6];
No
```

Because the lists we are using represent tokens present at certain places the order of elements is of no importance. However, if we use something like `select/3` we would receive the same tokens at least two times. (To be exact, we receive the same tokens n times, with n the number of elements in the target list). Because our search algorithm is constantly adding and deleting elements from markings, the search time would increase drastically. For every `add_marking` we would create *at least* two new branches. On the other hand, if we simply use the `append/3` operation, we avoid the introduction of useless branches in the search tree.

Why not a step with less rules ?

Wouldn't it be better to write a step with only one set of marking modifications instead of a step which, depending on which variables are bound behaves differently. The answer to this question is twofold. First, if both the input marking and output marking are unknown then the behavior is clearly different because we must assume that we are working with a relative marking (a marking only describing the necessary tokens), hence we start with empty token sets which will be filled up by the step predicate. The other case, when one of the input or output markings is bound requires also different behavior for every possibility:

- ⇝ if both the input and output marking are bound then we must simply verify whether the necessary tokens are present. We can do this by removing the elements from involved markings.
- ⇝ if only the input marking is known, we must *first remove the necessary tokens from the input marking* and afterward we can put the appropriate tokens in the output marking.
- ⇝ if only the output marking is known, we must *first remove the necessary tokens from the output marking* and then we can deduce which tokens should have been present in the input marking and put them there.

As can be seen, the order in which the tokens are investigated is important and different. This order is not strictly necessary, however, it greatly increases the performance of one step. Finding out what kind of tokens we could append to the unbound marking and then verifying whether these tokens can be found in the bound marking can easily lead to an infinite list of similar answers. Every new possible unbound variable might always be bound to the same token later on. We consider it better to get rid, as fast as possible, of free variables by actually binding them to one of the available tokens and then creating the required tokens for the unbound marking. By doing so, we a) increase the speed of the reachability analysis and b) get rid of a possible infinite amount of answers.

By implementing the step as we did, we do not break the declarative programming style. The step predicate can be used in any way necessary. For the user of `step` it behaves perfectly declarative. The only thing we did by introducing a verification of the boundedness of the variables is increasing performance.

8.6.2 Verifying Places versus Verifying Enabled Transitions

The concurrency module its responsibility is to bring the server component in a required state. More specifically, the required state is defined by the incoming $logic$message because every incoming $logic$ message must be accepted by the server. To make sure that the server keeps behaving correct, we restricted the possibilities of the enforce-action module to only interleave new $synchro$ messages. However, one problem was not anticipated, certain concurrency strategies require the ability to bring a server and its resources back to an old state. These servers typically embody a rollback mechanism. To allow this kind of logic, the reachability program must be modified to allow the use of $logic$ messages.

8.7 Summary

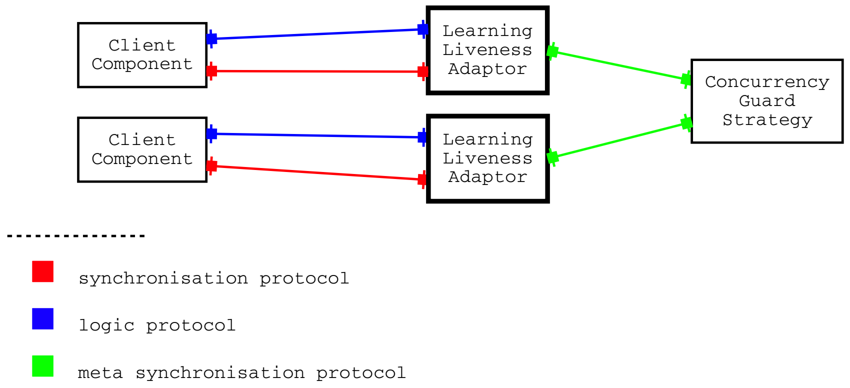
IN THIS CHAPTER we have shown how logic programs allow us to deduce easily how we can enable certain transitions. The process is fairly simply described in prolog but compared to standard formal reachability techniques it is fairly advanced:

- ⇝ It takes advantage of the colored Petri-net description. Instead of expanding a colored net to all its elementary places, the process works on a high level of abstraction by keeping variables as long as possible variable and unifying them only in the end.
- ⇝ It takes advantage of the possibility to reason in an abstract way about a marking. It does not require a fully described marking, it suffices to work only with the smallest necessary sub-markings. This results in a distinct advantage over other methods because all the possibilities introduced by non-relevant tokens are removed.
- ⇝ It reduces the search space drastically by doing a forward trace as well as a backward trace at the same time.

With this information, we can, given a petri-net description, a current marking and a target marking, easily deduce which actions should be taken to either a) enable a certain transition or b) place a certain token at a certain place. This enables us to create an adaptor that effectively bypasses the concurrency interface at a component, because all incoming actions will always be executed, no matter what state the component is in.

Chapter 9

Module 2: The Liveness Module



Client Component
Client Component
Learning Liveness Adaptor
Learning Liveness Adaptor
Concurrency Guard Strategy
synchronisation protocol
logic protocol
meta synchronisation protocol

Figure 9.1: The keep-alive adaptor

IN CHAPTER 7 WE have explained that we will construct an adaptor based on three requirements. Every requirement will be satisfied by one module. The enforce-action (explained in chapter 8) module allows the adaptor to satisfy the no-conflict requirement. The liveness module, presented in this chapter, will learn how to keep a client component (i.e., the one that requires a concurrency interface of the server component) alive. This is necessary to avoid concurrency adaptors from returning messages to the client that do not allow the client to proceed. E.g.: always returning `lockFalse`. To achieve this, the module makes use of a reinforcement learning algorithm that learns which actions are suitable in a certain situation. More specifically, actions are indirectly rewarded by the client component when suitable situations arise. How the rewards are assigned will be explained in section 9.2. The reinforcement learning algorithm will be linked together with a situation recognition system that allows for the storage of the necessary information to learn what to do. This situation recognition system will be Petri-net based. By constantly adding new random transitions and letting the reinforcement learning algorithm explore these new avenues, the correct rules will survive. This will be explained in section **9.1**. Finally, in section 9.3, we show how everything maps to a Q-learning algorithm.

9.1 A Petri-Net Representation

IN ORDER TO BE ABLE TO recognize which representation was best suited for our purpose we use genetic algorithms in a non-conventional way. Normally a genetic algorithm uses a certain representation to find a solution to a certain problem. However, in our experiments we used the genetic algorithms to test whether a particular representation is suited. To this end, we provided a number of problems the solutions of which were already known in advance. We checked how well the genetic algorithm performed on these problems. Most importantly, the representation we are looking for should lead to a solution for all presented problems. Moreover, the number of generations needed to find a suitable solution should be as low as possible.

During our experiments, that are explained in detail in chapter 11, we compared three different representations that seemed to be suitable: single-message classifier systems, multiple-message classifier systems, and Petri-nets. Only Petri-nets yielded satisfactory results. Therefore we will now focus on the Petri-net representation of our possible solutions.

9.1.1 Requirements for a Petri-net Representation

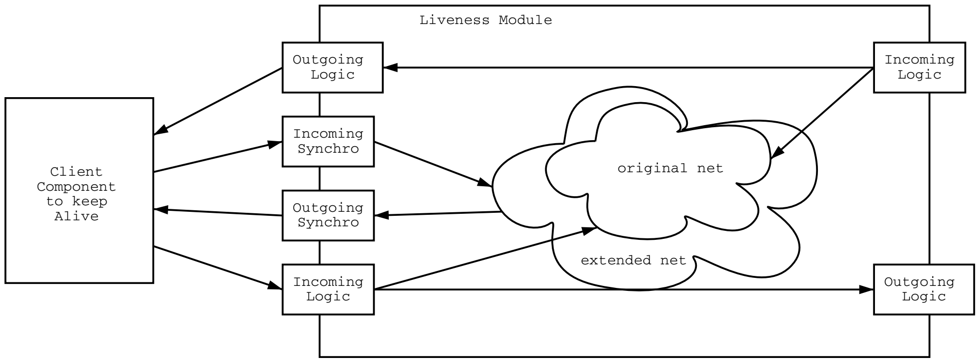


Figure 9.2: Schematic of the interconnection between the component, the adaptor ports and the underlying Petri-nets.

The goal of the liveness module is to keep the client component alive in a non-intrusive way: messages dealing with the core functionality of the client component (i.e., *logic* requests) are simply forwarded by the module to the server component. Synchronization messages, however, can be used by the module to alter the synchronization behavior of the client component. This is visualized in figure 9.2: On the left side of the figure is a client component located. It communicates with the liveness module by means of two interfaces, one *logic* interface, another *synchro* interface. Every interface has been divided in two parts. One part for incoming messages, another part for outgoing messages. The liveness module itself is connected to the remainder of the adaptor by means of one *logic* port, again pictured as an incoming and outgoing part. Internally, the liveness module will make use of two Petri-nets. One Petri-net describing and tracking the behavior of the client component, and a second net describing the behavior of the module.

From a more detailed point of view, the liveness module will continuously extend the existing client Petri-net with extra random transitions to introduce new synchronization behavior. The fitness of these random transitions will be taken into account automatically by the learning algorithm and as such only transitions that act correctly, will survive. However, one should take care that the newly added transitions do not interfere with the original behavior of the client Petri-net. For example, the extended Petri-net may enable some of the original transitions that could never be enabled in the original Petri-net. An extended Petri-net could allow a `SetPosition` when the position has not been locked yet. This is clearly undesired behavior. In general, the newly added transitions should never invalidate the preconditions of the original transitions

We will denote the extended Petri-net as $\pi_e = (\Sigma, P_e, T_e, A_e, C_e, G_e, E_e, I_e)$, while the original client Petri-net is denoted by $\pi_o = (\Sigma, P_o, T_o, A_o, C_o, G_o, E_o, I_o)$. Obviously, π_o must be a subnet of π_e. This means that $P_o \subseteq P_e$, $T_o \subseteq T_e$, $A_o \subseteq A_e$, $C_o \subseteq C_e$, $G_o \subseteq G_e$, $E_o \subseteq E_e$, and $M_o \subseteq M_e$ where M_o and M_e are the initial markings obtained from I_o and I_e, respectively.

The requirement that π_e does not interfere with π_o is formally expressed as follows:

$$\nexists M \text{ reachable from } M_e \text{ under } \pi_e \;:\; \neg(M \cap TE_{MS}^{\pi_o}\, reachable\, from\, M_o\, under\, \pi_o) \tag{9.1}$$

The definition of the multi-set of token elements $TE_{MS}^{\pi_o}$ was given on page 49 (equation 3.1). The reachability definition was given on page 65 (equation 3.5).

9.1.2 Runtime Creating of Transitions

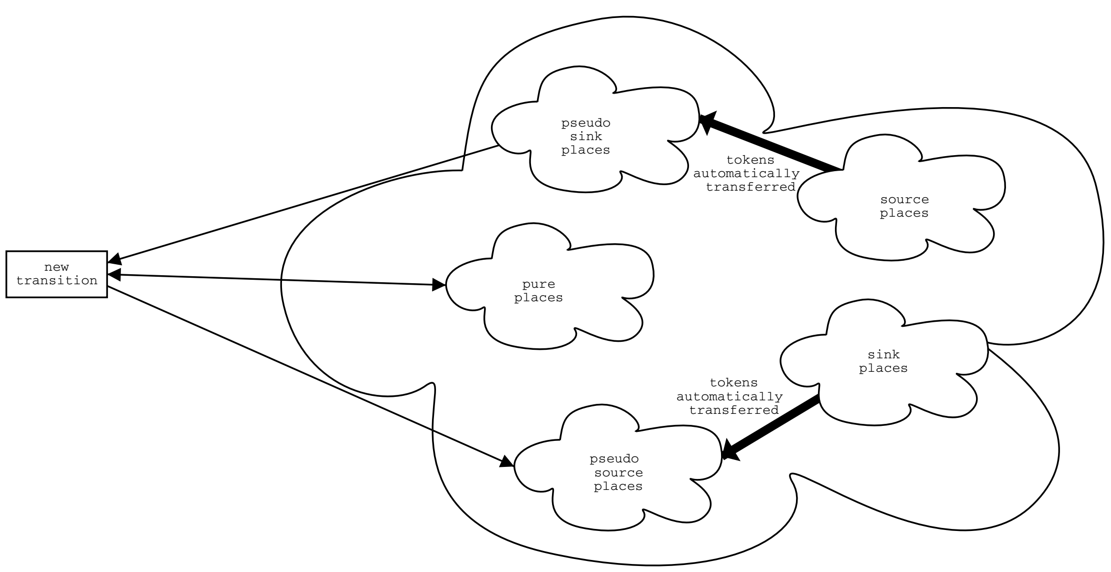


Figure 9.3: How new transitions will be created. The big cloud at the right is the original Petri-net. The new transition is part of the extended Petri-net.

At runtime, the liveness module knows which markings occur in the original Petri-net π_o. This knowledge allows us to optimize the process of generating random transitions. If a situation occurs in which the original Petri-net fails to take action, the module generates a new transition specifically suited for the corresponding marking. This is called **adding new behavior**. Otherwise, a random marking from the past is selected and the transitions related to that marking are modified at random. This is called **modifying behavior**. In both cases, every new transition generated by the module is inserted in the extended Petri-net π_e.

To **add new behavior**, we need to analyze the current (runtime) situation. This is done by retrieving the current marking and removing all source-places. The reason behind this is that a source-place can never be part of the *state* of a component, because it represents an incoming message. The second reason for ignoring the source-places is that every source-token is automatically transferred to a sink (or pseudo-sink) place. So if we need the information that a token is waiting we can as well look at the out-place instead of the in-places.

Removing all source-places yields a new marking that represents the (expected) state of the client component and the messages that cannot be handled (these reside in the pseudo-sink places). From this new marking the random generator selects two places that will form the inputs for the new transition. One of both places must belong to the original Petri-net, the other one must be a pseudo-sink place. (Illustrated in figure 9.3). To guarantee the non interference requirement (equation 9.1), new transitions must restore all consumed tokens back to their original places in the original Petri-net. Determining possible output places for the new transitions is also delicate. In the same way that we cannot allow a token to be removed from the original Petri-net, we cannot add new tokens to it. Therefore, only pseudo-source places can be used as output places. Of those, the only suitable candidates are those that enable a transition that sends back a synchronization message to the client component.

The expressions placed on the arcs that connect the input places to the transition and the transition to the output places is mainly based on the available type information. This idea comes from [Mon93, HWSS95], who argues that type information is an advantage when faced with random generated syntax trees.

To **modify the behavior** of an existing transition we simply keep the current input-arcs but modify the output arcs and expressions in the same way as above. The original transition is removed and replaced by the freshly created transition.

The representation given in this section is the best representation we found to represent solutions for our liveness problem. However, this representation now needs to be combined with an on-line learning algorithm. Such algorithms typically require feedback, for which we did not yet explain how we would obtain it. Therefore we will now turn our attention to the feedback problem.

9.2 Runtime Feedback

9.2.1 Check-pointing the Component's Source

THE PROBLEM WITH LIVENESS IS, as explained in section 7.2.3, that it is difficult to verify. In the formal Petri-net model liveness can be defined, but this wouldn't turn out to be a good measure because we don't actually know what the goal(s) of our components are. Therefore we need some feedback from the component that quantifies its liveliness.

A technique suitable to do so is check-pointing. A checkpoint is a static place in the code which will give rise to a reward when the code at that line is executed. Algorithm 25 illustrates this. By counting the occurrences of every checkpoint we can deduce a fitness measure. This fitness measure is specific for the component under investigation. Checkpoint (1a) will normally be reached only once. Checkpoints (1b) and (1c) will occur multiple times and should correlate in some way, because every point removed must have been once ours. Checkpoint (2) is a position which is a situation we don't want to encounter too much, hence when obtaining a quantitative measure we will reward this less than the other checkpoints. The green checkpoints (3a,b,c) measure how many lock operations are issued. By correlating the green checkpoints with the blue checkpoints we also have a measure for the underlying concurrency interface. This approach has a number of advantages and disadvantages.

Pros

- ⇝ The component can relatively easily indicate when it is alive and when not.
- ⇝ Non-determinism in the components can enhance the learning cycle by quickly removing 'hard'-coded behavior
- ⇝ The rewards will always be strongly correlated to the actual working of the component.
- ⇝ This form of documentation does not require much maintenance.

Cons

- ⇝ It might not always been possible to place rewards in the component's source. Especially not in open distributed systems. Therefore, In section 13.8.2, we investigate other possibilities. However, for now, we do assume that such rewards are offered by the component.

9.2.2 Correlation

In this section we will focus on an important property of checkpoints and Petri-nets, namely that the checkpoints reached are statistically correlated to the marking of the Petri-net. Later on we

Algorithm 25 Checkpoints in the floodactor algorithm. The green checkpoints are simple locking operations. The red checkpoint is a position we don't want to reach too often. The blue checkpoints are important 'mile-stones'. Every blue checkpoint has to be reached otherwise the component is not tested entirely.

```
(1a)     int color = joinActor();
         List seeds = new Vector();
         List border = new Vector();
         List ismine = new Vector();
         Pos pos = Pos.random();
         seeds.add(pos);
         while(seeds.size()>0)
           {
             int r=random.nextInt(seeds.size());
             pos = (Pos)seeds.remove(r);
             if (!lock(pos))
               {
(2)              seeds.add(pos);
                 continue;
               };
(3a)         if (!ismine.contains(pos) && isFree(pos))
               {
(1b)             setPosition(pos,color);
                 ismine.add(pos);
                 border.add(pos);
                 seeds.add(pos.left());
                 seeds.add(pos.right());
                 seeds.add(pos.up());
                 seeds.add(pos.down());
               }
(3b)          unlock(pos);
              Iterator it=border.iterator();
              while(it.hasNext())
                {
                  pos=(Pos)it.next();
                  if (ismine.contains(pos.left())
                   && ismine.contains(pos.right())
                   && ismine.contains(pos.up())
                   && ismine.contains(pos.down()))
                      {
                        lock(pos);
(1c)                    setPosition(pos,0);
(3c)                    unlock(pos);
                        it.remove();
                      }
                }
           }
```

will need this property, and although not the case in all situations we will assume that in most practical situations there is a statistical correlation between both.

We want to show that there is a statistical correlation between the state of the Petri-net and the checkpoints reached. So, if we know the marking of the Petri-net we should be able to give a probability that a certain checkpoint will be reached. It is clear that this question can be exactly answered *if* we have all information available. If we would have available the entire state of the component, then we can with a probability of 1 say which execution will be executed. To know this we can simply simulate the execution of the program.[1]

However, if we don't have all information available, this might not be possible. It is clear that the information contained within the Petri-net is only enough to describe the concurrency strategy of the component. It does not contain a full internal description of the state of the component, hence we might in general not be able to link these statistically. Typically, the checkpoints in components that cannot be statistically linked to the Petri-nets involved will use some extra information not available to the Petri-net. This information will be either information with respect to the concurrency strategy, or information with respect to some internal logic of the program. If the unknown information is part of the concurrency strategy, this should have been described in the Petri-net, if it isn't however, the information will still be in correlation with the concurrency strategy, because these checkpoints are specifically placed at important positions with respect to the concurrency strategy. Therefore we will further on assume that the checkpoints are statistically correlated to the marking of the Petri-net:

Assumption: the checkpoints help documenting the concurrency strategy in such a way that a statistical correlation exists between the checkpoints and the marking of the Petri-net.

In this section we have explained how rewards can be created by placing checkpoints in the component's source code. This is easy to implement and allows us to offer a reinforcement learning algorithm the necessary feedback.

9.3 Reinforcement Learning

THE PROBLEM OF KEEPING A COMPONENT ALIVE by selecting the correct action in a certain situation seems to be a suitable problem for a reinforcement learning approach as described in section 4.4. The mapping of our problem to a reinforcement learning problem requires the definition of states s_t, rewards r_t and actions a_t. We will define the state to be the marking of the Petri-net adaptor. The actions a learner can take are defined as the transitions within the net. However, part of the transitions are under control of the learner while others are executed automatically by the underlying component. If the action executed is not controlled by the learner it behaves exactly as if the learner would have chosen it.

$$s_t = M_t$$
$$A(s) = A(M) = \{a \mid M[a\rangle\} \tag{9.2}$$

The rewards for the learner come from the underlying component and are assigned at the moment a checkpoint in the component's code is reached. Because a reinforcement learning approach requires a reward signal at every time-step we define the rewards r_t to be zero unless the underlying component specifies otherwise. The underlying component will however send a reward for the last action back as an asynchronous message. This means that the learner must wait to assign a reward of zero until a new message arrives.

This definition of our liveness problem as a reinforcement learning problem is straightforward, however; before we can be sure that this problem is a valid reinforcement learning problem we need to be sure that the problem constitutes a Markov Decision process.

[1]One might object and say that we cannot be sure that the program will ever stop because it is Turing-complete. However, we *know* that it will stop because the components themselves are *reactive* and will finish always at some time.

9.3.1 A Markov Decision Process

We will now show that the stated liveness problem constitutes a Markov decision process. A Markov decision process is a process in which all necessary information from the past, that has led to the current situation is retained in such a way that a decision can be taken solely on the base of the current situation. Formally, $P_r(s_{t+1} = s', r_{t+1} = r')$ should be known[2] based only on the current state s_t and the current action a_t.

To demonstrate this we show that the current marking of the Petri-net and the current transition selected within the net define a) the next state and b) the reward that is returned. Proving a) is easy because the Petri-net describes *exactly* in a *consistent* way the concurrency behavior of the underlying component. Exact means that the next state s_{t+1} is entirely dependent on the current state s_t as defined by the Petri-net. Consistent means that the underlying component will act upon what is described in the Petri-net. Formally,

$$(s_t = M_t)\,[a_t\rangle\,(M_{t+1} = s_{t+1})$$

So,

$$\begin{cases} P(s_{t+1} = s') = 1 & s_t[a_t\rangle s' \\ P(s_{t+1} = s') = 0 & \text{otherwise} \end{cases}$$

Showing b), that the reward r_{t+1} is dependent on the current state s_t and the action a_t chosen is in general more difficult because the underlying component can make use of a memory which is not specified in the Petri-net, and which is thus not visible to the learner. However, if we go back to the assumption that the checkpoints are correlated to the Petri-net (page 148) then we can conclude that a reward is immediately dependent on the currently selected action and the current state of the Petri-net. Therefore we conclude that, under the stated assumption, the given problem is Markov.

9.3.2 Q-learning

Finding out which branch of reinforcement learning algorithms is useful (Monte Carlo, $TD(\lambda)$, Dynamic Programming) highly depends on the nature of the problem. The liveness problem we have is a *continuous* task because we cannot require the underlying component (and all the components it's communicating with) to reset their behavior during execution. This prohibits us from using techniques such as Monte-Carlo methods [SAG98]. The liveness problem also has no model available to foresee when a reward will be assigned in the future, therefore *dynamic programming* [SAG98] techniques are also of little interest to us. As such, we will investigate the use of *temporal difference learning*. A temporal difference learning algorithm approaches the problem of estimating the $Q(s,a)$ and $V(s)$ value-functions in an incremental way. Every time a certain action a_t is selected the resulting state s_{t+1} (or state-action couple (s_{t+1}, a_{t+1})) will give a certain amount of reward to the current state s_t (or state-action couple (s_t, a_t)). The next state is responsible for recreating its amount of accumulated reward by choosing an appropriate action. As such, rewards are propagated backwards. If future rewards are accessed multiple times, the current situation will always profit of it. Below we will investigate how the well known $TD(\lambda)$ learning technique Q-learning can be applied.

Q-learning in general works as follows:

1. Choose action a from s using a policy (ϵ-greedy for instance) and the Q - value function.

2. Execute action a, observe r and new state s'.

3. Change $Q(s,a) := Q(s,a) + \alpha(r + \gamma max_{a'} Q(s',a') - Q(s,a))$. With $0 \leq \alpha \leq 1$. A high value for α will make the learner learn quickly. $0 \leq \gamma \leq 1$ is the discount factor. This models the fact that future rewards are worth less than immediate rewards as explained on page 74 (equation 4.1).

[2] P_r is the probability distribution

4. The new state becomes the current state. $s = s'$

9.3.3 State Space Compaction

The problem with implementing Q-learning is that both methods needs to keep a memory of $Q(s,a)$. In our case however this is not feasible because the number of possible markings is very large. The reason behind this is that Petri-nets can express in a compact way concurrent processes, hence, even for small Petri-nets the number of markings can be very large. However, we will argue that a straightforward state space compaction is possible by only assigning strengths to transitions enabled under a certain marking. This will be a good consistent choice that will respect the structure of the problem involved. Below we will first assume that we have an elementary Petri-net, describing the behavior of an interface (see section 3.4.3). Afterward we will see how this naturally extends to colored Petri-nets.

Definition of $V(s)$ and $Q(s,a)$

When working with a reinforcement learning algorithm we have to define how "good" it is to be in a certain state and how "good" it is to select a certain action in a certain state. These two functions are called the state value and the state-action value functions. For our Petri-net the state value function is easily defined as the best future reward possible by any action enabled under that state, hence we define

$$V(s) := max(\{0\} \cup \{Q(s,a) \mid a \in s^\bullet\}) \tag{9.3}$$

This definition is useful because it states that the maximum reward that can be obtained from state s is either 0 or the maximum of the reward to be expected from all *enabled* transitions under s. The state-action value function $Q(s,a)$ can be deduced immediately from this definition because the execution of a will always result in state $s[a\rangle$ for which we know the expected future reward by the state value function. Hence,

$$Q(s,a) := V(s[a\rangle)$$

In the above two mutual dependent definitions, a marking must be remembered for every possible action. This is expensive because the possible markings can be very large. Therefore, we will now define an expected reward solely on the messages being send between the components. In the definition given below, $Q(a)$ denotes the strength of transition a.

$$Q(s,a) := Q(a) = strength(a)$$

Implementing the Q-learning algorithm as such, is straightforward

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma\, max_{a'} Q(s',a') - Q(s,a))$$

$$Q(a) = Q(a) + \alpha(r + \gamma\, max_{a' \in s'^\bullet} Q(a') - Q(a))$$

$$strength(a) := strength(a) + \alpha(r + \gamma\, max_{a' \in s'^\bullet} strength(a') - strength(a))$$

With this definition we only have to store strengths in transitions, we don't have to remember possible rewards for every (s,a) couple. We will now explain that very often this high compaction ratio forms no problem. Given our definition of $Q(s,a)$ and the assumption that all updates to $Q(s,a)$ aim to make a better approximation of the possible future reward, we will indicate that $Q(s,a)$ indeed defines the expected future reward.

Although a seemingly trivial statement, the real problem comes from the fact that $Q(s,a)$ only stores one strength for every a and not for every couple (s,a). So an update to $Q(s,a)$ will very likely modify $Q(s', a)$ too , which not necessarily means that an update to $Q(s', a)$ is appropriate. To demonstrate that an update to $Q(s, a)$ will never result in an inappropriate update of another couple $Q(s', a')$ we will demonstrate that when $a \neq a'$ the update does not interfere and that when $s' \neq s \wedge a' = a$ such an update will interfere, but the result is still what one would expect. In the case that $a' \neq a$ proving non-interference is trivial because $Q(s, a) = Q(a)$ and this strength is stored in the transition a itself. So this update does not modify $Q(a')$, and thus does not modify any $Q(s', a')$. In the other case when $s' \neq s \wedge a' = a$ an update on $Q(s, a)$ can only occur when a was enabled under s. Therefore a modification to $Q(s', a)$ will only occur when a is also enabled under s'. If that is the case this inference is not a problem because the expected reward under marking s' should also become the reward of $Q(s, a)$ because there is a means by which this possible higher reward can be obtained. With this we have shown that our definition of $Q(s, a)$ will always indicates what kind of a reward we can expect in the future. Below we give an example of a situation where $Q(s, a)$ and $Q(s', a)$ with $s \neq s'$.

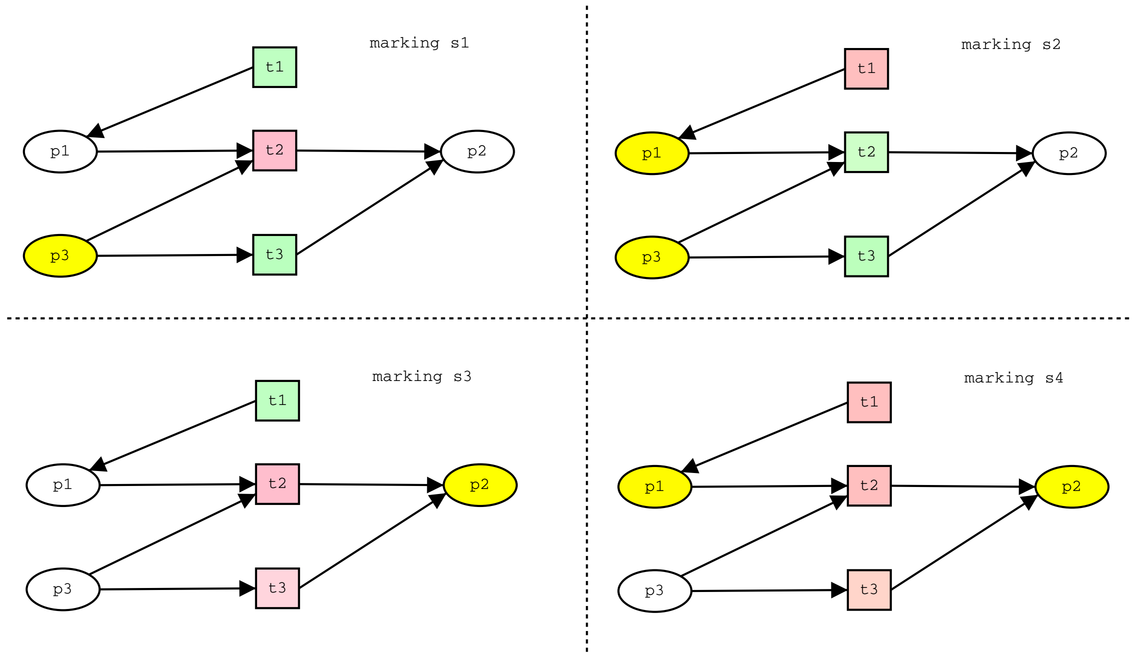


Figure 9.4: Four Petri-net states. When a place contains a token it is colored yellow. Red transitions are disabled, green transitions are enabled.

Example An example of such a situation is pictured in figure 9.4. We consider the case where two *different* markings both have the same transition enabled. In the top left part of the figure we see marking s_1. In the top right part of the figure we see a Petri-net with a current marking s_2. Both markings enable transition t_3. Marking s_4 (= $s_2[t_3\rangle$) receives a reward, but marking s_3 (= $s_1[t_2\rangle$) does not. In this case, after choosing action t_3 (executing transition t_3), a reward will be assigned to t_3 only in the bottom right case. This might indicate that an interference between $Q(s_1, t_3)$ and $Q(s_2, t_3)$ is in place. However, as explained earlier, the Petri-net marking is correlated to the rewards received, so we can assume that the reason why a reward is assigned for s_4 is because place p_1 contains a token, while it does not contain a token in s_3 (in this particular example). So t_3 will not preserve this information and not link state s_2 to a positive reward and state s_1 to no reward. Instead t_3 will oscillate around a certain value. The Q-learning algorithm will back propagate rewards to any transition fired before. Hence, after a learning period, transition t_1 will contain a constant value, indicating the necessity of a token at place p_1. This illustrates how the necessary information is preserved.

The oscillation that occurs on transition t_3 comes from the fact that the compression of all states (which actually constitutes a Markov Decision Process) loses too much information and no longer represents a Markov decision process. To be able to handle these unwanted situations, together with the need to handle specific symbolic actions (such as when [X, Y] arrives, send back [Y-1,X-1]) we have combined the above reinforcement learning algorithm with the situation recognizer described in section 9.1.

9.3.4 Structural Aspects of this Compaction

In the case of colored Petri-nets the above definition of $Q(s,a)$ will lose however even more information. For instance it is possible to have a set of differently colored tokens that all enable the same transition in different ways. Therefore, our definition of $Q(s,a)$ loses even more information, because a colored Petri-net typically collapses states when appropriate (as observed in the experiments in section 11.5). Our learning algorithm will be able to make use of this structural information. E.g. instead of learning what to do on position $(5,5)$, position $(6,8)$, position $(9,10)$ and so on, the learner will learn what to do on position (x,y). To do so, the learner will add new transitions every once in a while, depending on the policy. Furthermore, when necessary, the loss of information in $Q(s,a)$ can be compensated by creating new transitions that recognize new situations. Once such a transition is added its strength will increase or decrease, depending on its suitability.

9.4 Discussion

9.4.1 Experiments

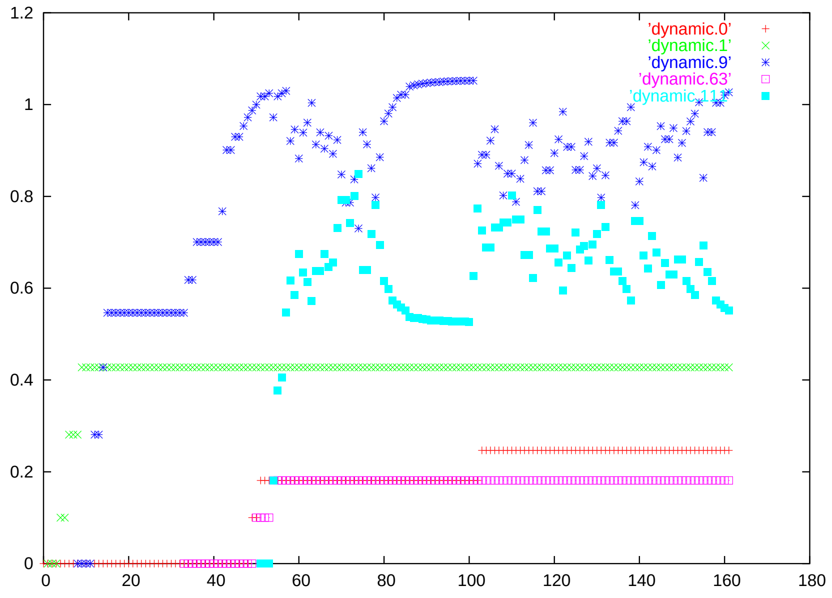
TO VALIDATE WHETHER THE COMBINATION of Petri-net transitions, reinforcement learning and statically placed checkpoints works in practice, we conducted a number of experiments. Every experiment is based on the conflicts enumerated in chapter 6.

We now describe the behavior of our reinforcement learning algorithm by looking at some of the more interesting experiments. The experiments performed apply a learning rate α of 0.1 and a discount factor γ of 0.9. Every experiment is summarized in a figure that plots the Q value of a number of transitions over time. Not all transitions involved in every liveness module are presented, only those that helped in implementing a correct behavior. For every transition the Petri-net code is shown under the form of *dynamic* transitions, together with the number of times the transition has fired during the experiment. Every line in the graphic has its own behavior, and specifies a transition that can only be fired in certain circumstances. For instance, in figure 9.7 we see 2 lines. The red line (dynamic transition 0) is the locking behavior, the green line (dynamic transition 1) is the unlocking behavior. Now, some observations can be made:

1. Often transitions will be present with a fairly constant Q value, such as the dynamic transitions 0 and 63 in figure 9.5. This typically occurs when the transition is very specific and only works on specific positions such as row 11 and row 12. Such transitions are therefore not often executed, hence the Q value is not adapted as fast as the often used transitions.

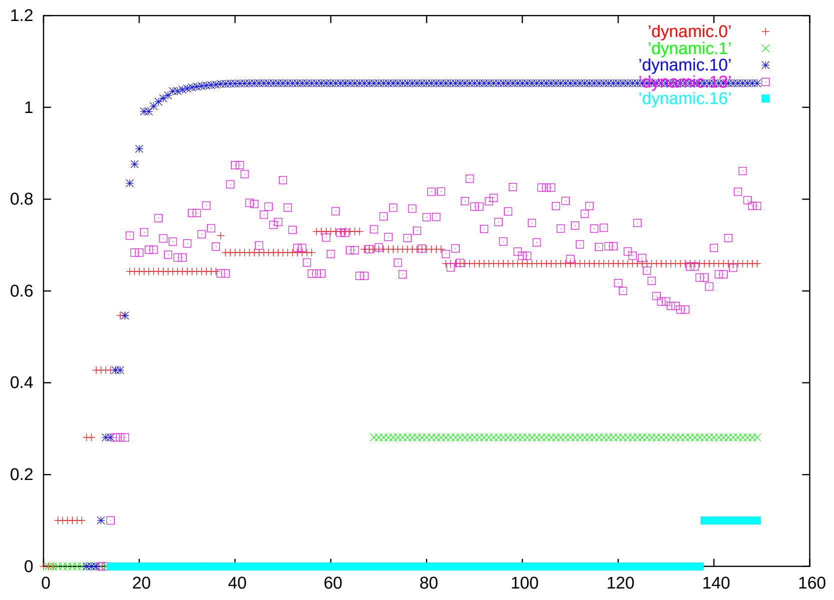
2. Sometimes transitions occur which share behavior. For instance in figure 9.6 the 16th dynamic transition implements exact the same behavior as the 10th dynamic transition. The state-recognizer generator currently avoid duplicates as much as possible, however in this case the names of the temporary variables are swapped and as such are not recognized as duplicates. However, such duplicates (dynamic transition 16) does not immediately take away the rewards from the original transitions (dynamic transition 10).

3. In figure 9.5 the expected Q-values of the two most used transitions oscillate. This comes from the fact that flood actor 1 not only assigns rewards when it is able to lock or unlock a square, but also when a position can be claimed to be part of the actor, in that case a



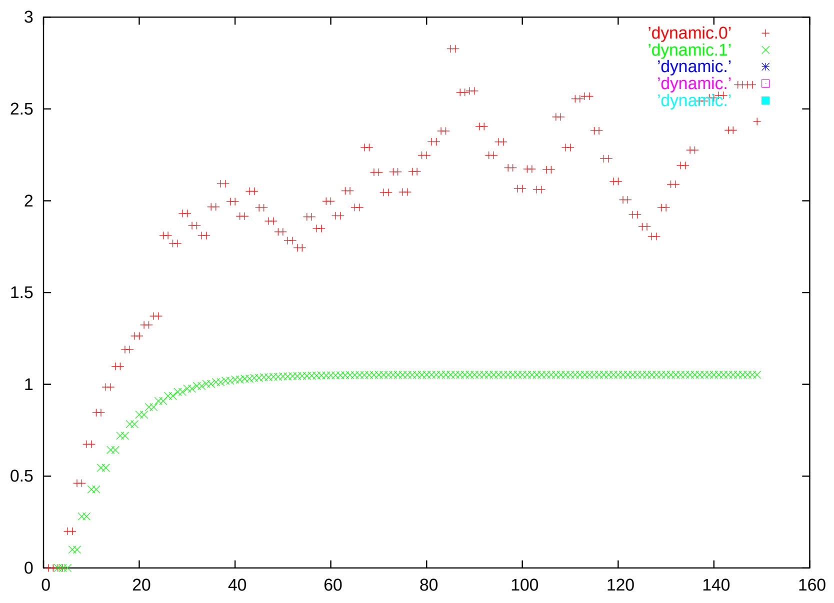
```
(transition "dynamic-0" (7 times)
  (input "lock-out"-req [TMP2 11])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
(transition "dynamic-1" (4 times)
  (input "unlock-out"-req [3 TMP2])
  (input "return_unlock"-enabled [])
  (output "return_unlock"-act []))
(transition "dynamic-9" (196 times)
  (input "unlock-out"-req [TMP2 TMP1])
  (input "return_unlock"-enabled [])
  (output "return_unlock"-act []))
(transition "dynamic-63" (4 times)
  (input "lock-out"-req [TMP1 12])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
(transition "dynamic-111" (175 times)
  (input "lock-out"-req [TMP1 TMP2])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
```

Figure 9.5: Liveness module tested on floodactor 1 (non-waiting, nested, square, immediate transition).



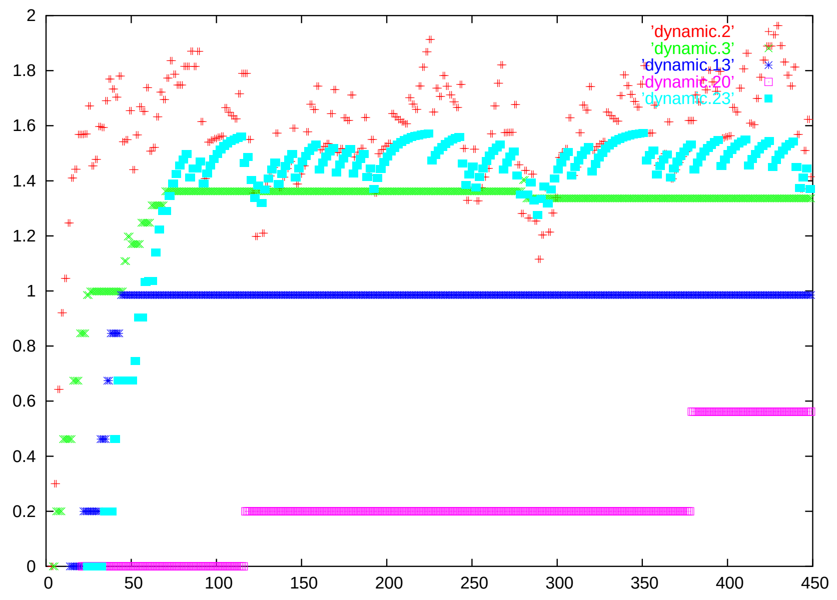
```
(transition "dynamic-0" (12 times)
  (input "lock-out"-req [9 TMP1])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
(transition "dynamic-1" (3 times)
  (input "unlock-out"-req [TMP1 14])
  (input "return_unlock_true"-enabled [])
  (output "return_unlock_true"-act []))
(transition "dynamic-10" (189 times)
  (input "unlock-out"-req [TMP1 TMP2])
  (input "return_unlock_true"-enabled [])
  (output "return_unlock_true"-act []))
(transition "dynamic-13" (182 times)
  (input "lock-out"-req [TMP1 TMP2])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
(transition "dynamic-16" (2 times)
  (input "unlock-out"-req [TMP2 TMP1])
  (input "return_unlock_true"-enabled [])
  (output "return_unlock_true"-act []))
```

Figure 9.6: Liveness module tested on floodactor2 (non-waiting, non-nested, squares, immediate, different syntax)



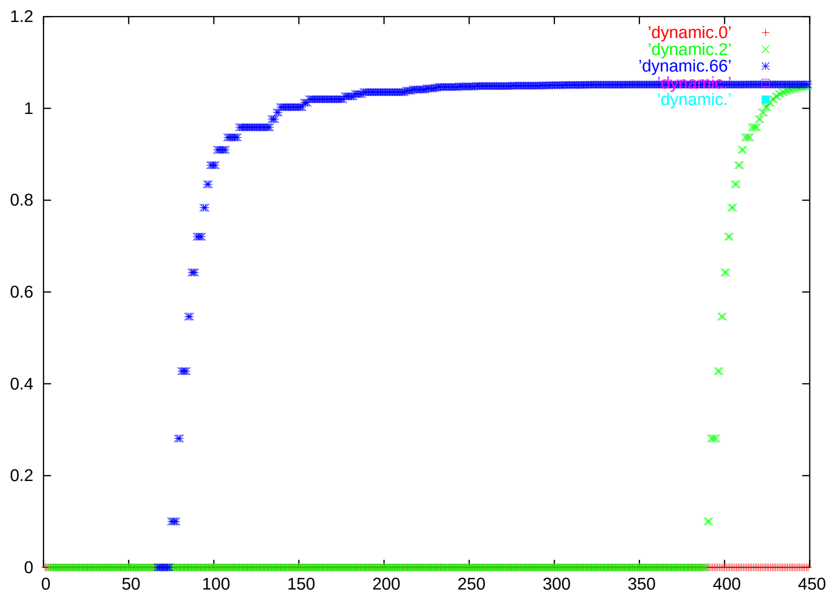
```
(transition "dynamic-0" **(74 times)**
  (input "enter-out"-req [])
  (input "enter_done"-enabled [])
  (output "enter_done"-act []))
(transition "dynamic-1" **(73 times)**
  (input "leave-out"-req [])
  (input "leave_done"-enabled [])
  (output "leave_done"-act []))
```

Figure 9.7: Liveness module tested on floodactor3 (waiting, non-nested field lock, immediate transition)



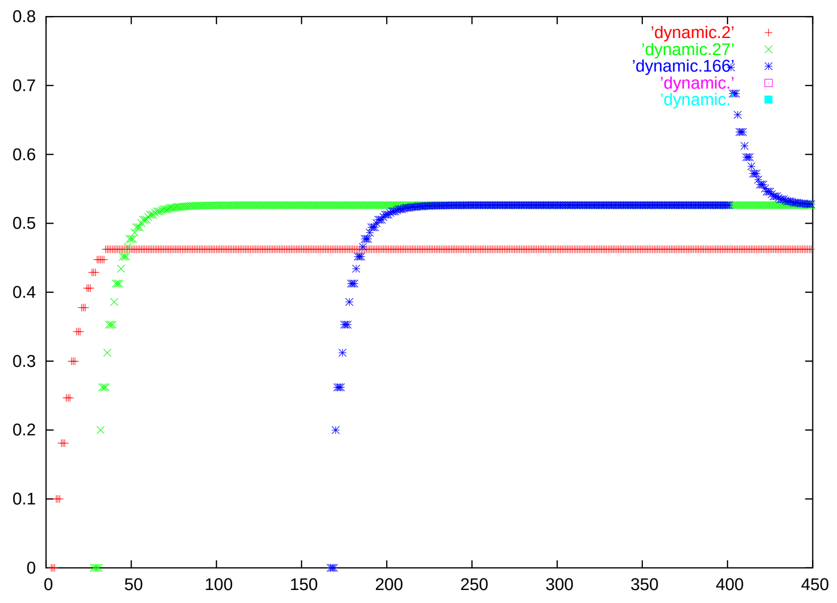
```
(transition "dynamic-2" (218 times)
  (input "lock-out"-req [TMP1 TMP2])
  (input "lock_done"-enabled [])
  (output "lock_done"-act []))
(transition "dynamic-3" (15 times)
  (input "unlock-out"-req [9 TMP1])
  (input "unlock_done"-enabled [])
  (output "unlock_done"-act []))
(transition "dynamic-13" (6 times)
  (input "unlock-out"-req [TMP2 8])
  (input "unlock_done"-enabled [])
  (output "unlock_done"-act []))
(transition "dynamic-20" (3 times)
  (input "lock-out"-req [TMP2 TMP1])
  (input "lock_done"-enabled [])
  (output "lock_done"-act []))
(transition "dynamic-23" (197 times)
  (input "unlock-out"-req [TMP2 TMP1])
  (input "unlock_done"-enabled [])
  (output "unlock_done"-act []))
```

Figure 9.8: Liveness module tested on floodactor 4 (waiting, nested, square, immediate transition, normal syntax)



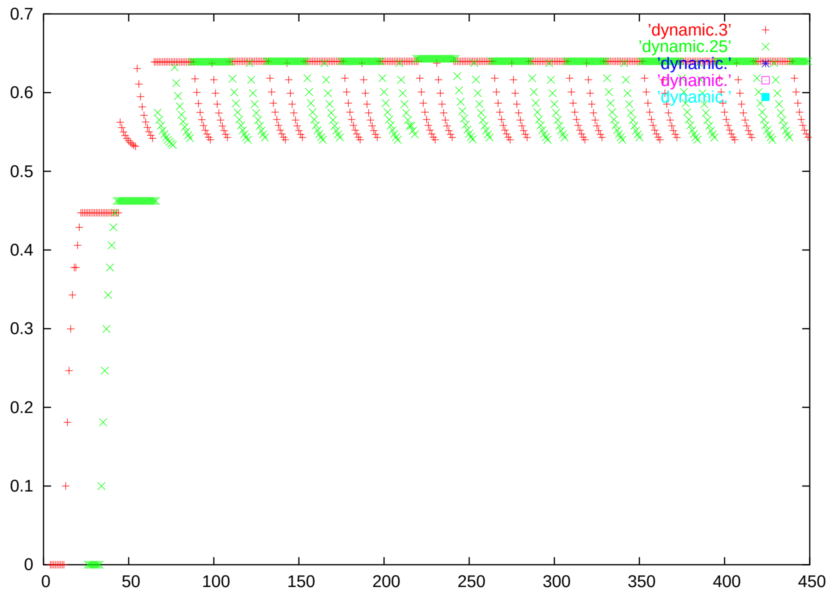
```
(transition "dynamic-0" (294 times)
  (input "lock-out"-req [TMP2])
  (input "lock_false"-enabled [])
  (output "lock_false"-act []))
(transition "dynamic-2" (27 times)
  (input "lock-out"-req [TMP1])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
(transition "dynamic-66" (39 times)
  (input "lock-out"-req [18])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
```

Figure 9.9: Liveness module tested on floodactor 5 (non-waiting, nested, line locking)



```
(transition "dynamic-2" (11 times)
  (input "unlock-out"-req [28 3])
  (input "return_unlock"-enabled [VAR0])
  (output "return_unlock"-act [VAR0]))
(transition "dynamic-27" (210 times)
  (input "lock-out"-req [28 TMP2])
  (input "lock_true"-enabled [])
  (output "lock_true"-act []))
(transition "dynamic-166" (141 times)
  (input "unlock-out"-req [VAR0 TMP1])
  (input "return_unlock"-enabled [VAR0])
  (output "return_unlock"-act [VAR0]))
```

Figure 9.10: Liveness modules tested on moving dot actor 6 (non waiting, non nested, square locking)



```
(transition "dynamic-3" (215 times)
  (input "lock-out"-req [TMP1 TMP2])
  (input "lock_done"-enabled [])
  (output "lock_done"-act []))
(transition "dynamic-25" (206 times)
  (input "unlock-out"-req [TMP2 TMP1])
  (input "unlock_done"-enabled [])
  (output "unlock_done"-act []))
```

Figure 9.11: Liveness modules tested on line actor 9 (2 layered locking strategy. The first layer is a waiting, nested square locking strategy, the second layer is a waiting nested field locking strategy)

relatively high reward (10) is assigned. It is clear that this latter reward behavior does not conform to the stated requirement that rewards should only be used to specify concurrency behavior, however, aside from some oscillation this does not pose much problems. Figure 9.8 illustrates this even more clearly, here also the future reward after a lock operation keeps on oscillating because the rewards are positioned at different places.

4. In figure 9.6 one of the first rules discovered (dynamic transition 0) is a rule that allow locks on column 9, however it takes a long time to discover a general rule that also works on other columns of the whiteboard. Visually this could be perceived as a flood actor which, during its first 12 moves, migrates only vertically. However, when a general rule is found, it quickly spreads horizontally. A similar behavior is encountered in 9.9.

5. In figure 9.9, a reward-less lock operation is executed 294 times. This happens because another transition (which knows how to lock column 18) hides the lack of reward. This comes from the fact that the quality of a certain marking is the maximum possible reward that can be obtained in the future. Given the fact that our colored Petri-nets collapses different states, and does not make a distinction between column 18 and all the other columns, transition 0 will be used to claim the future reward. However, because our representation of state-recognizers is well designed, this forms no problem, only a delay.

6. Figure 9.11 shows an experiment in which we tested how future rewards are remembered and how fast they are forgotten. The actor involved is a line actor, which only assigns a reward at the moment an entire line could be locked or unlocked. As can be seen, the future reward is forgotten slowly and thus still remembered correctly at a later time to allow the algorithm to exploit this knowledge.

7. In figure 9.11 the locking strategy contains two layers. The first layer is the layer in which we need to specify that we will start or stop with locking operations. The second layer can be used to lock individual squares. Because the offered Petri-nets do not allow much choice with respect to what to do when a `StartLocking` request arrives, these transitions are not included.

8. In figure 9.10 the goal of the experiment was to learn that in 50% of the cases a `LockFalse` should be sent back and in 50% of the other cases a `LockTrue` should be sent back. Learning this behavior has clearly failed. Currently, the amount of reward that is expected to be received is marginal in comparison to what can be obtained. This can be seen in the quality value of dynamic transition 166, in which at time step 400, suddenly a high reward arrives because the algorithm has tried to send back a `lockFalse`. However, the reason for this high reward cannot be remembered and is quickly forgotten again. This is an example of a hidden correlation that cannot be learned.

Of all the components tested (16 concurrency strategies), 15 could be kept alive by the liveness module. The one failing is component pictured in 9.10. From these experiments, we can conclude that our liveness module works as expected, however, we can also conclude that it might be overkill and that it seems that most concurrency strategies are in essence very simple FSM's, in which we don't have to remember much of the previous history. We will come back on this issue in chapter 12.

9.4.2 Bypassing a Concurrency Strategy

The approach we have presented in this chapter uses a reinforcement learning technique as a means to solve the liveness problem. Because we cannot rely on the formal liveness definition, we introduced checkpoints as a means for the developer to specify which actions are favored. The technique presented here could also be used to learn how an adaptor can *reach* a certain state. Should we want to do this, the reward could be assigned at the moment a change in marking occurs that is closer to the target marking. If it is further away from the target marking, a negative reward (punishment) should be given. Implementing such a technique would be quite similar and it would allow us to bypass the concurrency strategy of a server component without the need to formally analyze the Petri-net.

9.4.3 Comparison with Classifier Systems and the BBA

The cautious reader may notice that there are a lot of similarities between the Petri-net transitions we use and classifier systems. Indeed: the messages in a classifier-system can be compared to the tokens present in the marking of a Petri-net system. In that case,

- ⇝ Both, classifier rules and Petri-net transitions use preconditions to specify when they can be fired. For classifier rules these are the classifier conditions, for Petri-net transitions, these are the input-arcs and possible guards.
- ⇝ Both have an output interface: when the preconditions hold, the classifier/transition can be fired and certain actions will be taken. The classifier-rule can post a new message, the Petri-net transition can place a new token.
- ⇝ A classifier systems contains a number of classifier-rules which can be executed in parallel. A Petri-net contains Petri-net transitions that can be executed in parallel. In both systems, removing messages/tokens is done within a critical section.

Aside from these similarities, both strategies were investigated in an off-line fashion, by means of a Pittsburgh approach (described in detail in chapter 11). This technique is inherently off-line and the genetic algorithm requires different episodes to verify the value of every individual. However, if we want to make this technique work on-line we might apply the Michigan-approach [LF93], in which the different classifier rules are considered to be competing individuals. To evolve such an on-line strategy and learn which rules are appropriate in which situations, the bucket brigade algorithm [Hol] (BBA) might be useful. Summarized, the BBA gives each classifier a strength. If a classifier is fired it passes on its strength to the ones that created the current situation. In the next step, the classifier will receive a reward 'from the future' if there is a reward that could be obtained.

Nevertheless there are some major differences which make the BBA useless in our case. First is the fact that the bucket brigade algorithm needs to know how specific a certain rule is. Depending on the specificity a higher bid can be placed. This is something that cannot easily be transposed to Petri-nets. A second limitation of the BBA is that delayed rewards are implemented by means of a *mean* future reward. This effectively means that a possible very good solution can be hidden behind a lot of bad *alternatives*. A third problem with the BBA is the initial bias which is often present when introducing new rules. Evolutionary speaking, new rules can never prove themselves in an environment where systematically the worst rules, regardless of their age, are removed first. This makes balancing the exploration/exploitation phase very difficult.

9.5 Summary

IN THIS CHAPTER WE HAVE EXPLAINED how we can construct a liveness module. First, we explained which representation we would use to recognize new situations, second we explained how rewards from the underlying component could be obtained. And third, we explained how Q-learning can be applied to the given problem. In this chapter we have developed a method to minimize the storage-requirements of the value function, needed for Q-learning. Thanks to the locality property of Petri-nets we only need to store $Q(a)$ instead of $Q(s,a)$ and by introducing state-recognizers as new transitions within the Petri-net we are able to exploit structural properties of the colored Petri-net involved.

Chapter 10

Module 3: The Concurrency Module

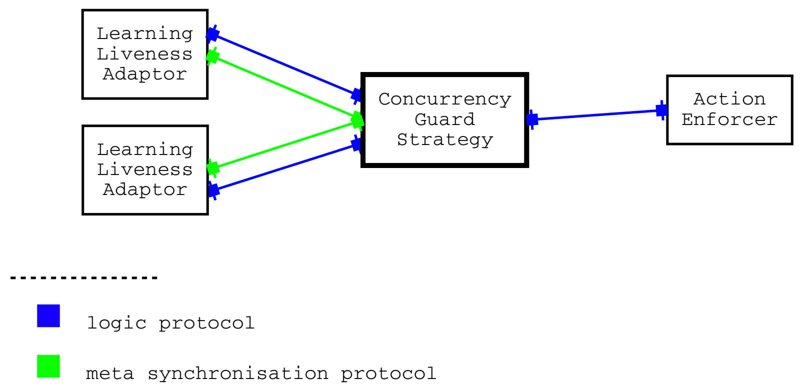

Learning Liveness Adaptor
Learning Liveness Adaptor
Concurrency Guard Strategy
Action Enforcer
logic protocol
meta synchronisation protocol

Figure 10.1: Schematic of the concurrency adaptor

UNTIL NOW WE HAVE ADDRESSED two sub-problems. These are a) the problem of enforcing an action on a component in such a way that the concurrency strategy is bypassed and b) the problem of learning how a component is best kept alive. The last module, which we will now present, is necessary to control access to resources by multiple clients.

Firstly we will explain how we can deduce which resources are involved, secondly we will explain what exactly the problem is we want to solve. Thirdly we will explain when a component has control over a certain resource. Fourthly, we will explain how our concurrency module can exert control over the resources involved. And finally we explain how race-conditions can be avoided.

10.1 Resources

BEFORE WE CAN EXPLAIN how we can manage access to the different resources we need to define which resources we will consider.

A resource in our case is defined as all the *logic* transitions that can be found in the elementary net, expanded from the colored Petri-net. Every such a transition is a resource in its own right. This approach to resources in a Petri-net is good enough for our case. Clearly the real resources will have a much larger granularity than the definition we impose here. For instance a resource in our whiteboard is at least the size of a square. If we consider resources to be the state of transitions then we have 4 times as many resources: a `setPosition(1,1)`, `returnSetPosition(1,1)`, `getPosition(50,70)` and `returnGetPosition(14,3)` for every square. However this is an advantage because it allows for a greater flexibility in aligning the resources. For instance, with our definition of resources one can specify that square x, y is locked for all operations and that square $x + 1, y$ is locked for the read-operations. Furthermore, this definition of resources is also complete in the sense that no resource can be found that cannot be expressed based on the state (enabled or disabled) of transitions. This definition can be extended to include the state of *logic* places of the core functionality. However we will continue to work with only the transitions.

Definition: A *resource* within our Petri-nets is defined as every possible logic transition in the elementary Petri-net expanded from the colored Petri-net.

With this definition of resources, we will now investigate what kind of race conditions can occur, and why we need to manage this.

10.2 The Requirement: What is a Race-Condition ?

THE NO-CONFLICT REQUIREMENT (given in section 7.2.1 on page 124) states that no action requested by the client should at any moment be impossible on the server. Simply by connecting the liveness-module to the enforce-action module we can easily guarantee this. However, if we do so, the no-race requirement is not satisfied. Below we illustrate why.

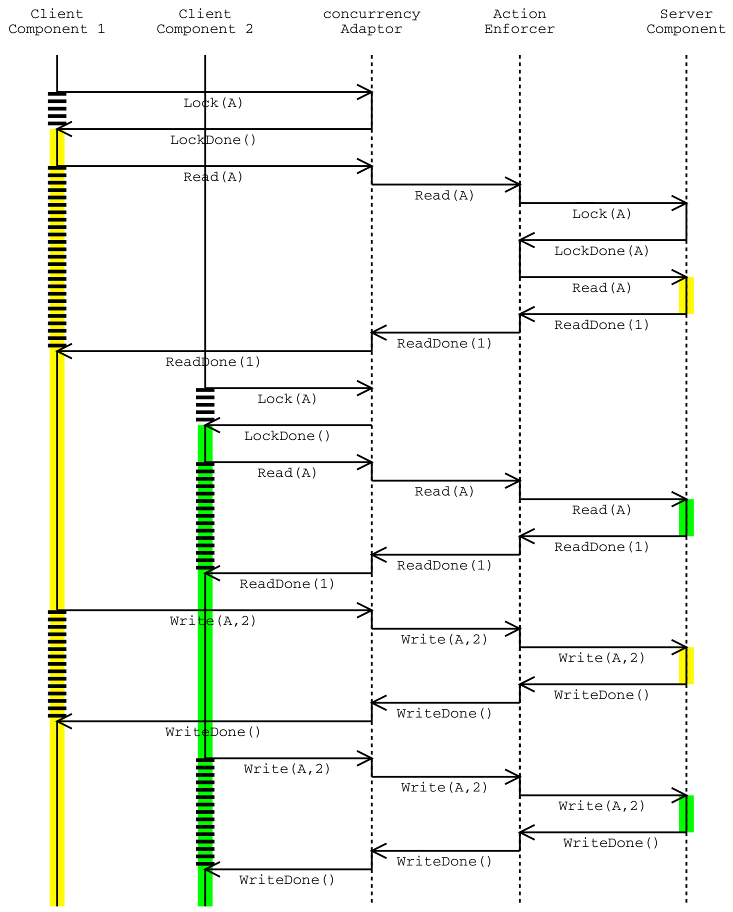


Figure 10.2: A race condition caused by the adaptor. The yellow and green lines are critical sections. As can be seen the critical sections interleave (race) on the server.

The reason why connecting the liveness module to the enforce-action module does not work is that we effectively remove *any* concurrency strategy on the server. As such, a client can execute any server action at any time it wants. This allows for a greater freedom, and effectively avoids the conflict. However, the server can no longer guarantee that a transition that is enabled for the first client is not reassigned to another client during execution. E.g., it is possible for an adaptor to unlock a resource locked by A, lock the resource for component B, unlock the resource for component B, and finally lock it again for component A. Here a concurrency problem will occur because component A expected the resource to be locked the whole time, while in practice another component was able to change the state of the given resource. This is pictured in figure 10.2.

To avoid this we should no longer only look at the fact whether a transition is enabled or disabled. The real problem lies with the *change* of enabled to disabled transitions not with the *state* they are in. If we want to make a concurrency module that avoids race conditions the concurrency module needs to understand when which component can change the 'enabledness' of a transition, or in other words when which component has *control* over which resource.

10.3 Control

In this section we investigate how resources can be controlled by the different clients. A resource is (as stated earlier) defined as the transitions within the expanded Petri-net. When a process has control over a certain resource it means that the process can access that resource and use it, or modify its state. In our case, we can easily define control over a resource as the ability to execute a transition, or modify the state of the transition. However, not all transitions of a Petri-net can be controlled by the underlying component. Most outgoing messages (messages such as `Lock`, `Read`, `Write`, `Unlock`) can be under control of the underlying component, however the incoming messages (messages such as `LockDone`, `ReadDone`, `WriteDone`) are typically not under control of the client component. This approach can also be found back in [MA97].

Definition: A transition is under control of a client component if it can be executed immediately or its state can be modified directly *or* indirectly through using *logic* messages.

Because the *incoming* and *outgoing logic* messages can be sent (or involuntarily) received, they take part in the process of verifying whether the state of a transition can be changed. Once an *outgoing logic* message is sent out, the concurrency module will simply pass it through to the server component. If an *incoming logic* message comes back from the server, this message is again passed through to the client component. On the other hand, *outgoing synchro* messages can be sent out by the client component, but do not need to be passed through and offer the concurrency adaptor a choice where it can choose which enabled *incoming synchro* message to send back. For instance, such a choice can be the request for a lock. The concurrency adaptor has the choice to accept or reject the lock by sending back a `LockTrue` or `LockFalse` message. However, both choices will result in different enabled/disabled *logic* transitions.

Once a `LockTrue` is sent back the client component can choose to execute a `Read` or `Write`, thus the read and write transitions are under control of the client component. The messages that can be sent back by the server component are `readDone` and `writeDone`, thus indirectly under control of the client-component. Given a Petri-net and a corresponding marking, this controlled-by information can be calculated automatically by means of a reachability analysis, similar to the one described in section 8.5.

10.4 Choice-Points

Before investigating how we can satisfy the no-race requirement we need to know what kind of actions can be taken by the concurrency module.

It is clear that the concurrency module can not simply insert new *logic* transitions, because this would alter the runtime behavior of the system. However, it should have the possibility to insert *synchro* transitions at certain places. However, inserting this kind of messages is already done by the enforce-action module and the keep-alive module. Nevertheless, the link between the concurrency module and the enforce-action/keep-alive module is still not fully explained.

Therefore, to be able to let the concurrency-adaptor choose a certain action, the keep-alive-adaptor will supply a set of possible *futures*, sorted according to their priority. This will allow the concurrency-adaptor to first try to reach a certain target future and acknowledge that future. If such a future cannot be reached without breaking the no-race requirement, another future will be tried until a suitable one is found. If no possible future can be realized the component will be set to wait until such a future can be realized.

From now on we will assume that the choices offered by the keep-alive module contains information as how to select a certain future and control-information that describes which states become under control of the given component when that future is selected. If a transition comes under control of the component we will mark it with a '+'. If a transition does not come under the control of the component we mark it with a '-', otherwise, if it is undecided we mark the transition with an '?'. We will write this down as an $1 \times m$ matrix, where m is the number of *logic* transitions in the exploded colored Petri-net. For instance

$$\begin{array}{c} \begin{array}{cccc} \text{clearPosition} & \text{joinActor} & \text{getPosition}(1,1) & \text{setPosition}(1,1) \end{array} \\ \left(\begin{array}{cccc} - & + & + & + \end{array} \right) \end{array}$$

Transitions for which it is undecided what the future will bring, will be considered to be under control of the client component.

10.5 Avoiding Race Conditions

GIVEN THE DEFINITION OF RESOURCES and the possibility to choose certain futures that extend different control over the resources, we can now explain how the concurrency module avoids colliding critical sections.

We will do this by means of two matrices. The first matrix, called the control-matrix is of size $n \times m$ where m is the number of resources available and n is the number of clients available. On this matrix the rows contains the clients. On the columns the transitions (resources) will be placed. If a resource is under control of a client a $+$ is placed in the matrix, otherwise a $-$.

The second matrix has the same layout but marks whether a certain resource is managed or not by the concurrency module. This is necessary for two reasons. First, a resource can always be enabled by different components (a resource like `joinActor` for instance) and second, sometimes certain clients don't have a concurrency strategy at all. It is clearly impossible to manage concurrency for a client that does not specify any concurrency strategy, or for resources that do not take part in a concurrency strategy. Therefore, the concurrency module needs to keep track of the resources and clients it can and will manage. If a resource/client is managed it is marked with a $+$, otherwise it is marked with a $-$.

$$C = \left(\begin{array}{cccccc} & joinActor & retJoinActor & getPos(1,1) & retGetPos(1,1) & setPos(1,1) \\ Client1 & + & + & - & - & - \\ Client2 & + & + & - & - & - \\ Client3 & + & + & + & - & - \\ Client4 & - & - & - & + & - \\ Client5 & + & + & - & - & + \end{array} \right)$$

$$M = \left(\begin{array}{cccccc} & joinActor & retJoinActor & getPos(1,1) & retGetPos(1,1) & setPos(1,1) \\ Client1 & - & - & - & - & - \\ Client2 & - & - & + & + & + \\ Client3 & - & - & + & + & + \\ Client4 & + & + & + & + & + \\ Client5 & - & - & - & - & - \end{array} \right)$$

The above example of the control matrix C and the managed matrix M illustrates the difference between both concepts. For instance column one and two can be a `joinActor` and a `returnJoinActor`, For all clients (except the fourth), this `joinActor` and `returnJoinActor` are under control of the client because the clients can choose to execute that transition at any time. However, the fourth actor, is unable to extend control over the `joinActor` because it must (for example) first call a `joinLocking` first. However, after the first `joinLocking` message, the concurrency module will recognize that the `joinActor` messages are effectively managed by the client component, hence the M matrix marks these resources and clients with a true. ($M[4, 1..2]$).

The columns 3,4 and 5 are the separate operations on all the squares off the board. The first and second client currently extends no control over the resources, however, only the second client actually makes use of a concurrency strategy for those resources ($M[2, 3..5]$). The fifth client on those resource has currently control over the last resource $C[5, 5]$, however, this control is unmanaged by the concurrency module $M[5, 5]$ and as such must be treated differently when realizing a certain future.

Therefore, we will now explain how $logic$ and $synchro$ messages are handled by the concurrency module.

10.5.1 Handling Incoming $logic$ Requests

The concurrency module handles incoming $logic$ requests by verifying whether the message is allowed for that component, given the current situation. If an action is allowed it can be passed through to the server component. If an action is not allowed, the concurrency module has no option but to place the action in a queue and wait until a suitable configuration of the other components arises to send the message through. An action can be allowed for two reasons. Firstly, it can be allowed because the action itself is managed by nobody, or the clients managing the resource currently does not extend any control over the resource. Secondly, an action can be allowed because it concerns a resource that is managed by the client-component and is controlled by the client. Formally:

A request t coming from client id is allowed $\iff$

$$C[id, t] \wedge M[id, t] \vee (\forall i \,|\, 0 < i \neq id < n \,:\, \neg(C[i, t] \wedge M[i, t])) \tag{10.1}$$

10.5.2 Handling Incoming $synchro$ Requests

A second kind of messages that should be handled by the concurrency module are requests to choose a certain future. These requests come from the client component only if there are multiple futures to choose from. The responsibility of the concurrency module is to choose a correct future that does not collide with the existing configuration of other clients. Therefore the C and M matrices will be used. However, three things need to be discussed. First we need to discuss how we can detect whether a resource is managed, second we will discuss when a future can be realized and third we will discuss what happens when such a future is realized.

In the discussion below we assume that a client with identification id has proposed a number of futures F_i $(i = 1..f)$. As already explained, a future is specified as a $1 \times m$ matrix, where m is the number of resources involved.

Detecting managed resources

A resource is managed at the moment a client component offers two different futures for that resource and enables the concurrency module to choose between both. Therefore, we will update the M matrix as follows

$$M'[i, j] = M[i, j] \vee \bigvee_{k<f,\, l<f} F_k[j] \neq F_l[j]$$

The above line states that a resource is managed if different futures have different control over the resource.

Is a future realizable ?

Given this information, we can now verify whether a certain future F can be realized for client id. In general, a future F can be realized if its resources can be realized. A resource j can be realized only if it realization does not block out *all* clients (including itself) to use the resource. To understand better how such a situation might occur, we investigate all the possible tracks. We do this by only looking at one resource j. If there exists one offending resource (in F) then the entire future is not realizable.

Intuitively, when future F specifies that it does *not* want to control resource j, then client id will not prohibit any other client from using the resource. This can be seen in equation 10.1. If $C[id, j]$ is false, then the term $\neg(C[id, j] \wedge M[id, j])$ will be true, hence not blocking other clients from using j. On the other hand, if F wants to control resource j then we must look at the state of the other clients. We now look for a client (denoted k) that might give rise to a situation that locks out all clients.

1. $C[k, j]$ is false. This will lead intuitively to no conflict because client k does not extend any control over the resource, hence will not collide in the future. If this is the case the module will *not* prohibit F from being realized.[1]

2. When $C[k, j]$ is true, we have a problem. Client id *wants* to lock resource j, but client k already has locked resource j. Now, depending on how the resources are managed different results may be obtained:

   (a) $M[k, j]$ is true. Here, resource k is managed and controlled by client j. Depending on whether id manages its resource, we have different possibilities

      i. $M[id, j]$ is true. If we would allow F from being realized then we would have two clients accessing the same resource. This cannot be allowed, hence we must *prohibit* F from being realized.

      ii. $M[id, j]$ is false. If we allow $C[id, j]$ to become true, client id would have no access to the resource as long as client k has locked resource j. Hence, this situation does not prohibit the realization of F.

   (b) $M[k, j]$ is false. Here, resource j is unmanaged by client k, but the resource is under control of that client. Depending on whether id manages the resource j, different situations can arise. If client id does manage the resource then client k will queue its $logic$ requests, while client id will be allowed to use the resource. Hence this does not prohibit F from being realized. However if client id does not manage the resource either, but still claims to control it then we will might have race-condition. Of course, this is to be expected because none of both clients manage the resource, so we should allow this behavior.

Formally we express this intuition as:

$$F \text{ can be realized} \iff \forall 0 < j \leq m : \neg F[j] \vee \bigwedge_{k \neq id} \neg collide(k, id, j)$$

$$collide(k, id, j) \iff C[k, j] \wedge (M[k, j] \wedge M[id, j])$$

[1] Please note that we say 'does not prohibit F from being realized' instead of 'allows F to be realized'. This is because we are discussing when *one* client k will block a future. It is perfectly possible that client k does not block a future, but that client $k + 1$ does so.

Realizing a future

One the concurrency module has determined which futures can be realized, it will choose one to realize. Realizing a future is easy. The only thing we need to do is copy F to the correct place in the matrix C. Thus:

$$C'[id, j] = F[j]$$

If no future can be realized the request is placed in a queue until it can be realized.

10.6 Discussion

BELOW WE DISCUSS THIS APPROACH. We investigate how certain clients behave when mediated by means of this module. We will explain that our module in most cases behaves as expected, however in certain special cases it will fail to work.

10.6.1 Server Requirements

The first observation about this module is that the concurrency strategy of the server does not matter ! The enforce-action module uses the concurrency strategy only to bring the server in a required state. The fact that we no longer need to take into account the offered concurrency strategy is important for two reasons. First, it is a strong result because not a single server component in such an automatically mediated conversation needs to offer a concurrency strategy. Second, it allows us to look only at the indirect conflicts between different client components that want to access the same server. This is what we will do now.

10.6.2 Unmanaged Concurrency

client 1	client 2	client 3	without adaptor	with adaptor
unmanaged	unmanaged	unmanaged	leads to race conditions	leads to race conditions
unmanaged	unmanaged	managed	leads to race conditions, for all clients	leads to race conditions except for client 3

We return to our conflict cases and investigate how our adaptor might improve the communication between multiple clients and a server. To do so, we first observe the behavior of a set of clients with no concurrency strategy. Afterward we observe the same set of clients for which only one requires a concurrency strategy.

If no client involved in a communication session specifies a concurrency strategy then the adaptor will simply pass through all requests. To understand this, we point out that, if no concurrency strategy is introduced, then no resource is managed, hence we end up in branch 2b (section 10.5.2). By passing simply passing through any request, the module will not avoiding race-conditions. Hence, the adaptor will mediate the concurrency conflict and will still exhibit the implicitly required behavior, that is, not to manage concurrency.

On the other hand, if one of the connected clients offers a concurrency strategy, all the unmanaged clients might need to wait until the concurrency managing client has finished its operations. To understand this we point out that a concurrency strategy will lead to managed and controlled resources. When one of those is present, all other clients will set to wait until the one managing its resources has finished.

This strategy clearly improves the working of the system. If the adaptor would simply mediate the differences, the unmanaged concurrency clients might access the server while the managed concurrency strategy is supposed to access resources exclusively. This observation shows how our adaptor might not only mediate, but also improve the concurrency behavior.

10.6.3 Livelocks

client 1	client 2	client 3	without adaptor	with adaptor
rollback	rollback	rollback	might lead to livelocks	can avoid livelocks

The presented concurrency module does not care about livelocks. For instance, if none of the involved components takes the possibility of starvation into account, then the resulting behavior may exhibit a behavior that will lock out a certain client. However, it is possible to insert a learning algorithm in the concurrency module that will learn how to avoid livelocks and learns which timing to apply to keep all involved components alive. Because this behavior was not part of the required interface, this is clearly an improvement. However, improving the working of the whole is not always be possible, such as in the case for deadlocks.

10.6.4 Deadlocks

client 1	client 2	client 3	without adaptor	with adaptor
waiting	waiting	waiting	leads to deadlocks	leads to deadlocks

If the involved components require a concurrency strategy which is deadlock-prone, such as a waiting square locking strategy, then the adaptor will not avoid deadlocks. The adaptor will still mediate the differences in syntax, control-flow, re-entrance, granularity conflicts and other variabilities. This shows that the adaptor working as required, however it will not resolve deadlocks automatically. It would be nice to have an adaptor that not only mediates the differences but also avoids any concurrency problems. In section 13.8.6 we will briefly explain how we could approach this. However, our adaptor will be able to detect a deadlock if one might arise.

10.6.5 Implicit Requirements

client 1	client 2	client 3	without adaptor	with adaptor
staircasing	staircasing	staircasing	no deadlocks	no deadlocks
staircasing	staircasing	waiting	leads to deadlocks	leads to deadlocks

Another observation we can make, is that certain implicit requirements cannot be detected, thus not mediated. Sometimes deadlocks can be avoided by cooperating clients if they apply a staircasing technique (see section 5.8.3 on page 94). If the clients apply such a technique then the adaptor mediates the differences and works as expected, thus avoid deadlocks. However, if another client, one that does not honor this implicit requirement, joins, then the whole conversation might deadlock. In section 13.5 we come back to this issue.

10.6.6 Virtual Resources

The only case where the concurrency module does not behave as expected is when certain resources are 'virtual', in the sense that they cannot be deduced from the Petri-net. Such a situation occurs for instance in conflict 6.12 (page 110). In this conflict the client requires the possibility to first lock the entire field. However, the Petri-net involved (picture 10.3) does not have any *logic* transition that indicates that the entire server should be considered to be a resource. Specifically, it is possible that the client requests to `enter` the server. This forms two problems. First, because the liveness module has no choice but return `enterOk`. Second, because if this future is involuntarily realized, none of the resources come under control of the client component. In this case the concurrency adaptor fails to work as expected. However, if the `enter` and `leave` transitions would be coupled to real *logic* transitions, then this would not form a problem.

From these results we can conclude, that if we present our adaptor with a set of concurrency strategies the result will work as expected and mediate the conflict correctly. On the other hand,

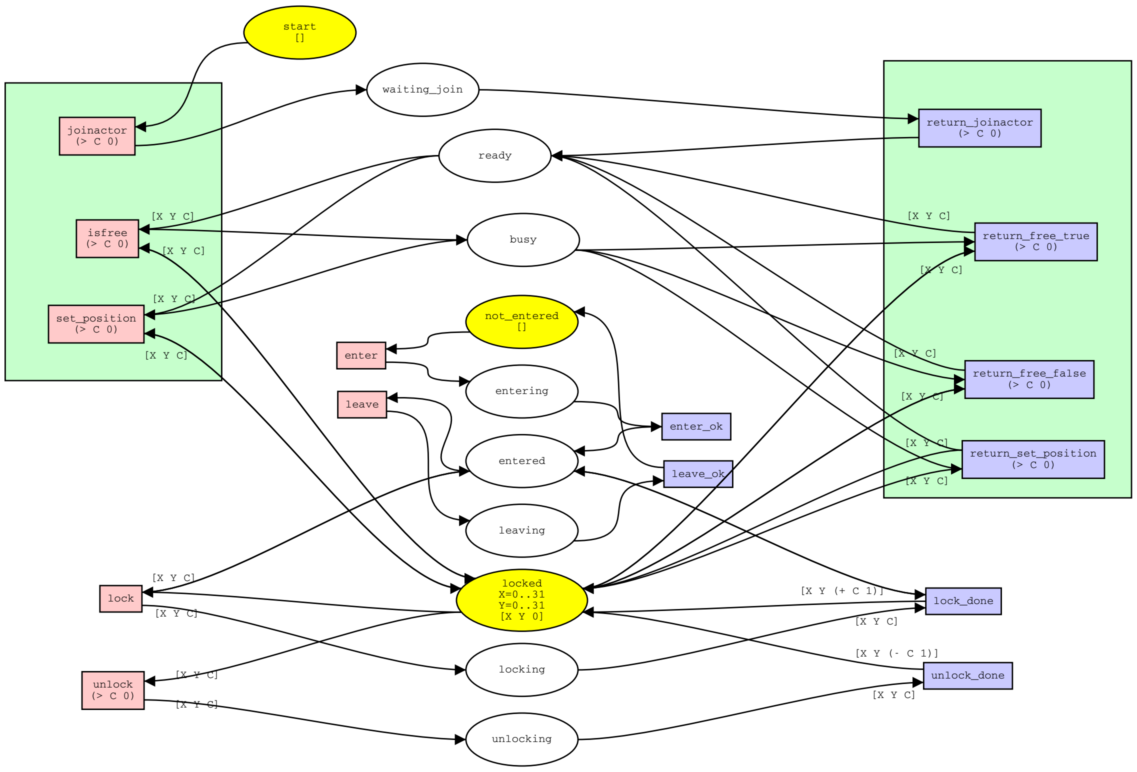


Figure 10.3: A Petri-net of a layered concurrency strategy. The green boxes are the logic transitions which form the resources involved.

if the input contains virtual resources or contains implicit requirements, the resulting behavior might be incorrect by introducing deadlocks. However, in every case, the adaptor will still mediate the conflict.

10.6.7 Implementation Issues

One of the implementation issues we encountered is a problem of performance. We define a resource as a transition within the expanded elementary Petri-net. This means that for a simple 32×32 whiteboard, the expanded Petri-net will offer around $32 \times 32 \times 5$ resources. Creating this expanded Petri-net takes a long time if not optimized.

Also, because the liveness module need to specify which resources they will control, they have to transmit relatively much information. To optimize this only change sets could be used. Also, certain resources are similar. If an act can be done on a resource, then a `getPosition` is also always possible. Deducing this information might help in optimizing this module, it might also help in better defining what a resource exactly is, given the Petri-net.

10.7 Summary

IN THIS CHAPTER WE HAVE EXPLAINED how we can easily create a module that avoids race-condition on the server. Firstly we explained which resources we consider. Secondly, we explained the requirement to avoid race conditions. Thirdly, we explained when a component controls a certain resource. This was necessary to be able to specify when a resource is involved in a critical section. Fourthly, we explained how our concurrency module is able to control the behavior of the clients involved and how it manages to avoid race conditions.

Part IV

Validation

Chapter 11

Experiments

We have a habit in writing articles published in scientific journals to make the work as finished as possible, to cover up all the tracks, to not worry about the blind alleys or describe how you had the wrong idea first, and so on. So there isn't any place to publish, in a dignified manner, what you actually did in order to get to do the work.

– Richard Feynman, physicist, Nobel Lecture, 1966

IN THIS DISSERTATION, until now, we have explained how we can create a on-line working concurrency adaptor. This adaptor, however has not been created over one night. A number of different kinds of experiments has been performed. These experiments have highly influenced the current construction of the adaptor. Therefore, we feel it is necessary to explain these experiments in more detail. In this chapter we will give an overview of them. In the following chapter we will discuss the adaptor we have presented in this dissertation.

11.1 Introduction

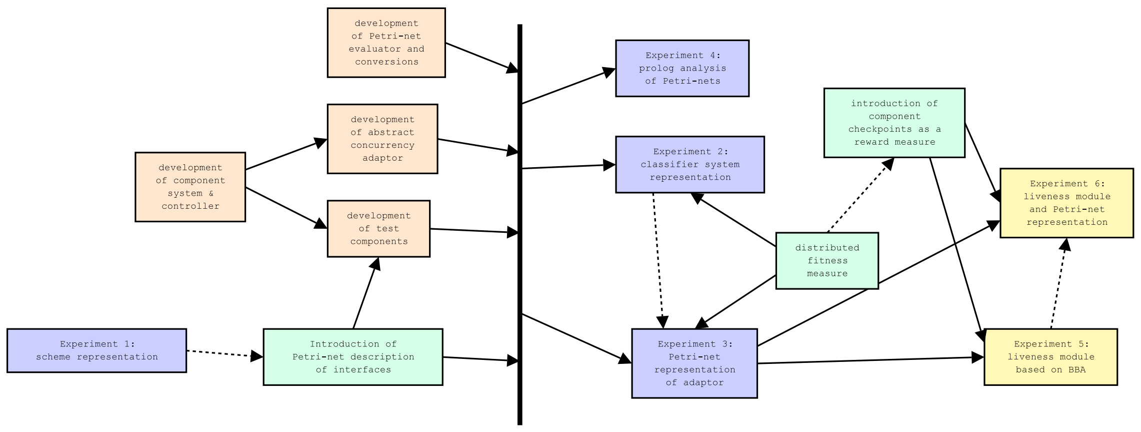

development of Petri-net evaluator and conversions
development of abstract concurrency adaptor
development of component system & controller
development of test components
Experiment 1: scheme representation
Introduction of Petri-net description of interfaces
Experiment 4: prolog analysis of Petri-nets
Experiment 2: classifier system representation
introduction of component checkpoints as a reward measure
distributed fitness measure
Experiment 3: Petri-net representation of adaptor
Experiment 6: liveness module and Petri-net representation
Experiment 5: liveness module based on BBA

Figure 11.1: Dependency graph of the performed experiments and developed tools. The orange boxes are development boxes. The green boxes represent the addition of extra information, such as Petri-nets, or component checkpoints. The blue boxes are off-line techniques. The yellow boxes are on-line techniques. Solid arrows are 'need'-dependencies. Dotted arrows are 'approach based on' dependencies.

PICTURE 11.1 CONTAINS A dependency graph of the experiments performed and the tools developed. First, we designed and implemented the SEESCOA component architecture together with a controller that can startup a number of components and connect them to each other. This has been done in Java. Second, we created our own Petri-net evaluator and accompanying tools (pretty-printer, conversion to prolog, conversion to .dia[1] files, conversion to low level nets). Third, we implemented a large variety of test components (as specified in chapter 5), together with a Petri-net representation of their concurrency strategy[2]. Fourth, we implemented a connection tracer that allowed us to verify whether the Petri-net was complete, hence that no message was not documented in the Petri-net. (described in section 3.6.3).

Below we will briefly summarize every experiment we have conducted. Initially we tried to create *one* adaptor to solve the concurrency strategy conflicts. However, after some experiments we took the decision to leave the track of off-line generated adaptors and focused on the practical usability of such an adaptor, hence we tried to make it work on-line. This posed some problems, which forced us to divide the adaptor in different modules as explained in chapter 7. The details of all experiments can be found on-line at http://progpc26.vub.ac.be/ cgi-bin/seescoacvs/component/concurrency/.

11.2 Experimental Setup

DURING THE EXPERIMENTS WE HAVE WORKED with two kinds of approaches. One approach was an off-line approach in which components are supposed to offer a test-component (pictured in figure 11.2). In such a setup, the learning algorithm has the opportunity to make wrong decisions which brings one of the participating components in an invalid and irreparable state. The second approach is an on-line approach in which a learning algorithm, or a learned algorithm, was supposed to work without failure. From both approaches, clearly the second is most realistic, because it can hardly be expected that in an open distributed system a component will offer 'test'-components, nor should any adaptor bring a component in an invalid state. However, as we have already explained in section 4.3.1, the off-line genetic approach offered some valuable insights into the representational problems we faced.

11.3 Measuring Technique

AS EXPLAINED IN CHAPTER 4, every learning algorithm needs some kind of feedback. If we make use of an off-line strategy to generate an adaptor we should be able to measure how well that adaptor performs (the fitness) to solve the problem.

Measuring the correctness of a generated adaptor is not at all easy because it can be a Turing complete process, which means that we actually have to try it and cannot even see when it has stopped, or is executing in a loop and doing something useful. If we would have enough processing power available we could try testing all adaptors in parallel to each other. If we assume that all concurrent processes run equally rapidly we could measure every generated adaptor over a virtually infinite time span. All we would need to do is test all individuals concurrently and summing up the rewards as they are received. When a certain adaptor has become the worst it is killed and replaced by a new adaptor. However, this way of working has a major drawback: it is immensely resource consuming, both computationally and with respect to the memory requirements. [DK96, GW93] Therefore we will need to fall back to another approach.

To solve this problem of measuring a possible Turing complete process we introduce the notion of *episodes.* (such as in section 4.4.2). An episode is a finite run of a certain adaptor, that is an execution with a limited number of messages that should be handled. The amount of fitness received after this finite time is the reward for that episode. At the beginning of an episode the

[1]This is the file format in which most of the pictures of this dissertation are represented.

[2]To avoid confusion: Petri-nets are used to document the behavior of a component, but they are also used in a Petri-net generator to create new behavior.

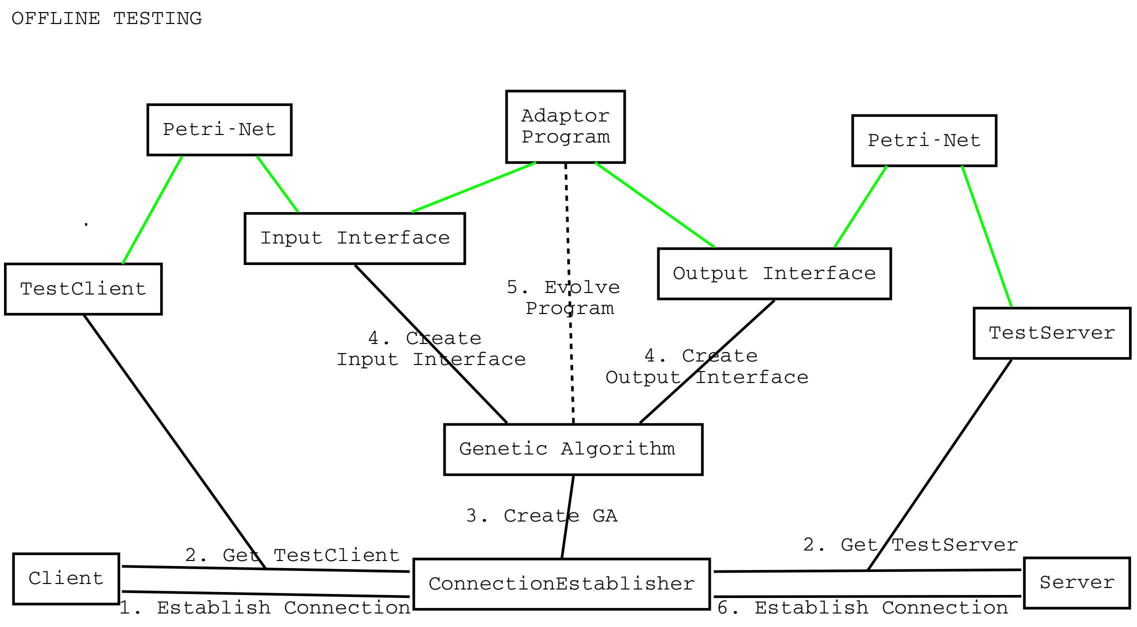


Figure 11.2: Setup of the offline strategy

fitness is always zero (so the fitness of an adaptor does not take into account previous runs of the adaptor.). If at the arrival of a certain message the Petri-net of the adaptor goes into a loop, we terminate it after 2 seconds[3] and assign the current accumulated rewards as a fitness. If handling a message takes longer than 2 seconds, the component is killed. This technique of working is well known when working with learning algorithms, however often evaluation-steps are measured instead of real time. [Koz92]

The running time for every episode is increased slowly as the number of generations increases. By doing so we not only solve the resource problem of testing all algorithms at the same time, but we increase a number of interesting properties:

- ⇝ With every new generation, every individual has to prove itself, thereby it can receive every time a better reward because the length of the episodes is increased. Therefore, any adaptor that has received once a good overall reward because it took a too specific action, will be removed very quickly in the following generations. For example: an adaptor that specifically works very well on position with $X-$coordinate 5, because the input expressions requires a 5 to be present at that place, will be removed in the following test round.
- ⇝ Because the length of the episodes is slowly increased, the search-algorithm can find out solutions to new problems every time a new problem is offered. For example: the search-algorithm will first try to solve the problem of the `Lock` operation. A while later it will need to solve the problem of the `Unlock` operation.

Below we will chronologically explain every experiment we have performed.

11.4 Experiment 1: A Scheme Representation

Goal: Creation of one adaptor program between two conflicting concurrency strategies.
Means: Genetic Programming with Scheme as a programming language.
Deployment: off-line

[3]2 seconds is an arbitrary time which is *more* than enough to let a component handle 1 message.

Input: *No* Petri-net description of the conflicting concurrency strategies is present.
Output: Scheme program that is supposed to mediate two conflicting concurrency strategies.

11.4.1 Description

In this small experiment we tested how a Scheme program [SJ75] could be generated automatically that would solve the conflict of a nested locking strategy at one side and a non-nested locking strategy at the other side. Essentially, this is a very easy problem to solve for an adaptor. The only thing it has to do is keep track of a counter, which specifies how many times the client has already locked the resource at the server.

The operations of the genetic algorithm are scheme operations such as `+`, `-`, `/`, `*`, `>`, `<`, `<=`, `>=`, `and`, `or`, `not`, `sendmessage`, `set!`, `define`, `if` and `handlemessage`. The smallest solution we are looking for is given in algorithm 26. In the program, the first argument given to `sendmessage` is the *port* to which to send the message to.

Algorithm 26 Solution to the nested client locking strategy and non-nested server locking strategy.

```
(define counter 0)
(handlemessage 'Lock' 0
  (set! counter (+ counter 1))
  (if (= counter 1)
    (sendmessage 1 'Lock')))
(handlemessage 'Unlock' 0)
  (set! counter (- counter 1))
  (if (= counter 0)
    (sendmessage 'Unlock' 1)
(handlemessage 'LockResult' 1)
  (sendmessage 0 'LockResult')
(handlemessage 'UnlockResult' 1)
  (sendmessage 1 'UnlockResult'))
```

11.4.2 Results

From this simple experiment we observed the following:

1. The random generation process was unable to create the above program. This is to be expected because the high number of basic operations, makes creating such a program very unlikely if done at random.

2. The random generation process highly influences the kind of generated programs. We could tune the process to generate this simple program, however, the algorithm would then be completely unable to generate anything else in other situations.

3. Measuring the fitness of an adaptor program is a discrete measurement, this means that the concept of 'gradually evolving to a correct working adaptor' is not at all easy.

4. The Scheme representation simply passes messages from one interface to another, it can barely handle the content of the messages, unless extra operators would be added, but if this were the case finding *one* suitable solution at all would become even more time consuming.

5. The lack of formal interface documentation was clearly observed in this program, because we had to supply the genetic algorithm with the terminals `lock`, `unlock`, `lockResult` and `unlockResult`.

11.5 Experiment 2: Classifier System Representation

parameter	value
individuals (genotype)	variable-length classifier system represented as bitstring
population size	100
maximum generations (100 runs)	11
parent selection	ranking selection (10 % best)
mutation	bitflip on non ranked individuals
mutation rate	0.8
crossover	uniform
crossover rate	0.1
input/output interfacing	Petri net state/transition representation
actions	message sending
fitness	number of successfully executed actions

Table 11.1: Parameters and characteristics of the genetic program

Goal: Create a concurrency adaptor between two conflicting concurrency strategies
Means: The Pittsburgh Approach to implement a genetic programming approach
Deployment: off-line
Input: Petri-net descriptions of two conflicting concurrency strategies & distributed fitness measure
Output: Classifier system to fill in the missing behavior

11.5.1 Description

Classifier systems (as described in section 4.3.2) are a well known technique to create algorithms by means of a genetic program. Because the Scheme representation suffered from too many problems, we changed our approach substantially. First we introduced a formal interface description under the form of Petri-nets, second we left the track of commonly used languages and investigated the use of classifier systems. In these experiments we were using the previous example of a non-nested versus a nested locking strategy. Both strategies offer simple enter/leave locking semantics. For a quick overview of the parameters of our genetic program we refer to table 11.1.

The goal of this experiment was to create a classifier system automatically (the Pittsburgh approach) that would mediate the differences between two conflicting concurrency strategies. The classifier system should reason about the actions to be performed based on the available input. Therefore we need to represent the external environment in some way and submit it to the classifier input interface. We have investigated the use of two different approaches to represent the external environment.

- ⤳ *Single message reactive systems*: in a single message reactive system, the classifier system receives one message that declares the whole external environment. This includes the message that comes in, the messages that can be sent out and the state of the underlying component.
- ⤳ *Multi message reactive systems:* in a multi message reactive system, the classifier system receives multiple messages. Every message specifies a piece of the environment. There are messages to describe incoming component messages, messages to describe outgoing component messages and messages that all together declare the whole state of the Petri-net.

Single message reactive systems

We now describe the semantics of a single message reactive classifier system.

The input of a classifier consists of the full state of the client and the full state of the server, together with the actions which are required by either one of them. So, at first sight we should create messages which contain the state in which the Petri-net is in and the action requested by the client or the server. Of course, this would allow our classifier system to take invalid actions in a certain context. The classifier could for example decide to send an act message to the server when the server wasn't even locked. To avoid this we decided to represent the client/server state by means of the possible actions either one of them can perform. We can easily obtain these because we have a Petri-net model that describes the behavior of all connected components.

In the case of simple enter/leave locking semantics, we define the semantics of our messages as a 28 bit tuple (shown in figure 11.3), where the first 7 bits are the possible server actions, the next 7 bits represent the possible client actions, the next 7 bits represent the requested server action and the last 7 represent the requested client action.

Server Possible Actions						
L	L#t	L#f	U	U#	A	A#

Client Possible Actions						
L	L#t	L#f	U	U#	A	A#

Server Requested Action						
L	L#t	L#f	U	U#	A	A#

Client Requested Action						
L	L#t	L#f	U	U#	A	A#

Figure 11.3: L = lock; L#t = lock_true; L#f = lock_false; U = unlock; U# = unlock_done; A = act; A# = act_done. A bit is set when either a certain action is possible or a certain action is required.

With these semantics for the messages, simply translating a lock from client to server requires three rules (see figure 11.4). One which transforms the client *lock* request into a server-lock call (rule 1) and two which transform a server *lock return* into a client *lock return*. (rule 2 and 3).

	Condition1	Action	Comments
1.	1###### ####### ####### 1######	####### ####### 1###### 0######	If we can call lock upon the server & the client wants to call it
2.	1###### #1##### #1##### #######	####### ####### #0##### #1#####	If we can call lock_true upon the client & the server wants to call it
3.	1###### ##1#### ##1#### #######	####### ####### ##0#### ##1####	If we can call lock_false upon the client & the server wants to call it

Figure 11.4: The bit patterns from left to right and top to bottom: server possible actions, client possible actions, server requested actions, client requested actions. A description of the bit semantics can be found in 11.3.

Although this is a simple example, more difficult actions can be represented. Suppose we are in a situation where the client uses a nested-locking strategy and the server uses a non-nested locking strategy. In such a situation we don't want to send out the lock-request to the server if the lock count is larger than zero. Figure 11.5 shows how we can represent such a behavior.

	Condition1	Action	Comments
4.	1###### 1##0### ####### 1######	####### ####### 1###### 0######	If the client wants to lock and has no lock we send the lock to the server
5.	####### 1##1### ####### 1######	####### ####### ####### 01#####	If the client wants to lock but has already a lock we immediately send back an lock_true.

Figure 11.5: Translating a client-lock to a server lock when necessary.

Multi message reactive systems

We now describe the semantics of a multi message reactive classifier system.

The message that resides in the input interface of such a classifier system contains a prefix that specifies what kind of message it is. When 000, the classifier message is about an incoming action from the client, 001 is an incoming action from the server, 011 is an outgoing action to the server and 010 is an outgoing action to the client. The prefix 10 describes the state of the client and 11 describes the state of the server.

The *rules* of such a classifier system consist of a ternary string representation of the client state and server state (as specified by the Petri-net), as well as a ternary string representing the requested Petri-net transition from either the client process or the server process. With these semantics for the classifier rules, translating a request from the client to the server requires only one rule. Another rule is needed to translate requests from the server to the client (see table 11.2).

The number of bits needed to represent the states of each Petri-net depends on the number of states in the Petri-net as well as the variables that are stored in the Petri-net (e.g., the `LockCount` variable requires 2 bits if we want to store 4 levels of locking).

classifier condition			action	rule description
requested transition	client state	server state	performed action	
00####	####	###	11#...#	Every incoming action from the client (00) is translated into an outgoing action on the server (11)
01####	####	###	10#...#	Every incoming action from the server (01) is translated into an outgoing action to the client (10)

Table 11.2: Blind translation between client and server processes. The last 5 characters in column 1 represent the corresponding transition in the Petri net. The characters in the second and third column represent the states of the client and server Petri net, respectively. The fourth column specifies the action to be performed based on the information in the first four columns.

Although this is a simple example, more difficult actions can be represented. Consider the situation where the client uses a counting-semaphores locking strategy and the server uses a binary-semaphores locking strategy. In such a situation we don't want to send out the lock-request to the server if the lock count is larger than zero. Table 11.3 shows how we can represent such a behavior.

classifier condition			action	rule description
requested transition	client state	server state	performed action	
00 001	~##00	###	10 010 ...	If the client wants to lock (001) and already has a lock (~##00) we send back a lock_true (010)
00 001	##00	###	11 001 ...	If the client wants to lock (001) and has no lock (##00) we immediately send the message through (001).

Table 11.3: Translating a client process lock request to a server process lock action when necessary.

In this experiment, which is an offline experiment, two components, with accompanying Petri-net descriptions of their behavior were offered to a genetic algorithm. The genetic algorithm would create complete classifier systems to link the two Petri-nets together. Initially, a single message reactive system was used, but as it turned out this classifier system was only able to learn very local behavior[4], therefore a multi-message reactive system has been tested. For the single reactive system, we will now define what our individuals are, how crossover and mutation

[4]That is, behavior which was too specific.

happens and what we do to measure the fitness. These choices are standard choices for classifier systems.

Individuals are initially empty. Every time a certain situation arises and there is no matching gene we will add a new gene. This gene, which is a classifier rule has a matching condition with a random action on the server and/or the client from the list of possible actions. This guarantees that we don't start with completely useless rules which cover situations which will never exist and allows us to generate smaller genes which can be checked faster.

Mutating individuals is done by randomly adapting a number of genes. A gene is adapted by selecting a new random action.

To create a *crossover* of individuals we iterate over both classifier-lists and each time we decide whether we keep the rule or throw it away.

Fitness is measured by means of a number of test-sets (which has already been discussed in section 13.8.2). As explained, every test-set illustrates a typical behavior (scenario) the client will request from the server.

The genetic programming uses a steady-state genetic algorithm, with a ranking selection criterion: to compute a new generation of individuals, we keep (*reproduce*) 10% of the individuals with the best fitness. We discard 10% of the worst individuals (not fit enough) and add *cross-overs* from the 10% best group[5]. It should be noted that the individuals that take part in cross-over are never mutated. The remaining 80% of individuals are *mutated*, which means that the genes of each individual are changed at random: for every rule, a new arbitrary action to be performed on the server or client is chosen. On top of this, in 50% of the classifier rules, one bit of the client and server state representations is generalized by replacing it with a #. This allows the genetic program to find solutions for problems that have not themselves presented yet.

The scenarios offered by the client are the ones that determine what kind of classifier system is generated. We have tried this with the test scenarios, as illustrated in figure 13.6. Scenario 1 is a sequence: [`lock(), act(), unlock()`]. In all scenarios, we issue the same list of messages three times to ensure that the resource is unlocked after the last unlock operation.

11.5.2 Results

requested transition	client state	server state	performed action	description
00 00 01	10 00 #1	11 ####	00 10 01	client?lock() & client=locked → client.lock_true()
00 00 01	10 00 1#	11 ####	00 10 01	client?lock() & client=locked → client.lock_true()
00 00 01	10 00 00	~ 11 010	00 11 01	client?lock() & client ≠ locked → server.lock()
00 00 10	10 00 1#	11 ####	00 10 11	client?unlock() & clientlock>2 → client.unlock_done()
00 00 10	10 00 01	11 ####	00 11 10	client?unlock() & clientlock=1 → server.unlock()
00 00 11	10 #####	11 ####	00 11 11	client?act() → server.act()
00 01 10	10 #####	11 ####	00 10 10	server?act_done() → client.act_done()
00 01 00	10 #####	11 ####	00 10 00	server?lock_false() → client.lock_false()
00 01 01	10 #####	11 ####	00 10 01	server?lock_true() → client.lock_true()
00 01 11	10 #####	11 ####	00 10 11	server?unlock_done() → client.unlock_done()
00 00 01	10 00 00	11 010	00 10 00	client?lock() & server=locked & client ≠ locked → client.lock_false()

Table 11.4: The generated classifier system for a single run.

An examination of the results of several runs of our genetic programming algorithm lead to the following observations:

[5] These values were taken from [Koz92] and gave good results during our experiments.

1. We found that (for all 100 runs of the genetic algorithm) a perfect solution was found within at most 11 generations. One of the classifier system that is generated by the genetic algorithm, when providing as input all test scenarios, is given in table 11.4. The produced classifier system simply translates calls from client to server and vice versa, unless it concerns a `lock` call that should not be made since the server is already locked. The bit patterns in the example differ slightly from the bit patterns explained earlier. This is because we need the ability to make a distinction between a 'transition-message' and a 'state-message'. All transition messages start with 00 and all state-messages start with 10 for client-states and 11 for server-states.

2. The best individuals were often created by only using mutation and creation of new genes. In the experiments we conducted we observed that the crossover operation is not necessary to find a suitable solution. This is because there are in essence not many parameters that work on the fitness independently. Another reason for this behavior is that the creation of new classifier-rules is based on a process that very actively selects a transition that is known to be enabled in a certain situation. Since most of the time not that many messages can be received or sent by a component, it is quickly checked in multiple generations which action is the best action in a certain situation.

3. About 50% of the created adaptors do not manage concurrency, but shortcut the missing behavior. That is, they offer the client a feedback behavior (such as the liveness module), without ever contacting the server.

4. *The Classifier Representation of Petri-nets is Too Low Level:* A problem with single message classifier systems is that their representation loses a lot of useful information. For instance if a rule has been found to lock one square in a clean way, it is not at all easy to extend this very specific rule to be a more general applicable rule. Unless the representation is very specifically suited to do that. This is especially a problem for the single message reactive classifier system, because the classifier system would need to invent an appropriate adaptor-strategy for *every* single square. This means that we have 1024 times more work than for one square. Above all it is not guaranteed that we will find a suitable strategy for all possible squares, because test-runs might not access all squares systematically. A solution to this problem would be to change a crossover operation and generalize one working set of rules to another set of rules, thereby taking into account the fact that repetitions occur at certain places. This information is available in the Petri-net, however, it is lost within the classifier system.

5. *Classifier Systems have too Low Level Operators:* In another experiment we tested the multi-message representation. After investigating this representation it became clear that it also was not suitable to do the job. This representation would allow the genetic program to find a suitable conversion strategy that could translate all incoming `lock` request to outgoing `lock` request in a way that would be automatically generalized. To allow this the coordinate of both the incoming and the outgoing lock-request was encoded within the bitstream at the same position. Nevertheless, if there would be a slight shift within the representation of incoming or outgoing `lock` requests (e.g: the incoming x of a `lock` is located on bits 3 to 11 and the outgoing x is located on bits 5 to 13), then it would become nearly impossible to find satisfactory classifiers. In general, the problem with classifier systems is in general that their basic operations are *very* basic, there is not even a notion of a variable.[6]

[6]To stress this point even more: consider the implementation of an addition operation in a classifier system. To do so, the classifier system needs to increase a binary coded number, by one. Writing down a classifier which does exactly this turns out to be very difficult, because often more than one bit needs to be flipped. To understand this, think of writing a set of general classifier rules that translates the bit pattern '01111111', which represents the decimal number 127, to '10000000' (the decimal number 128), such that is also suitable for other binary represented numbers. A solution to this problem exists and is called grey-coding [KRCE01]. Here numbers are represented in such a way that the hamming distance between two grey-coded numbers, x and $x + 1$ is exactly 1.

In general these results lead us to conclude that classifier systems offer too low level operations and are unable to exploit structural properties of the presented interfaces. These problems are recognized by [SS89]. Therefore, we investigated another representation: adaptors with an internal Petri-net representation. By doing so we hope to take advantage of a) the structural knowledge present in colored petri-nets and b) make use of better high level operations.

11.6 Experiment 3: A Petri-Net Representation

Goal: Create a concurrency adaptor between two conflicting concurrency strategies
Means: Genetic Programming using Petri-nets as a representation
Deployment: off-line
Input: Petri-net description of two conflicting interface & a distributed fitness measure
Output: Petri-net that links the different interfaces

11.6.1 Description

In this experiment, which is again an off-line experiment, another representation was chosen. To avoid the problems of the classifier systems, we now create a high level Petri-net description that will be measured at runtime. The runtime measurement is still distributed and contains two parts. On the server part, it keeps track of how many errors occurs and warnings that happened. On the client side the fitness is measured at how many locks could be obtained. The representation itself is already detailed in section 9.1.

11.6.2 Results

Component	# generations	Best Fitness	Transitions	Comments
BallActor1	10	38	2	
FloodActor1	23	80	2	
LineActor1	36	122	2	
BallActor2	25	161	2	
FloodActor2	26	150	2	
LineActor2	59	700	2	
BallActor3				
FloodActor3	30	250	2	
LineActor3				
BallActor6	13	20	2	a
FloodActor6	30	190	2	
LineActor6				
BallActor9				b
FloodActor9	20	140	4	b
LineActor9	30	20	2	a,b

Table 11.5: Experimental results of the priority-first searching algorithm, using the petri-net representation described in the text.

The results of this experiment were promising. First, we noticed that this representation had no problem at all to find out transitions to offer a correct 'mediated' behavior. Furthermore we noticed,

1. that random aspects within the components behavior quickly leads to correct solutions, while static behavior often leads to more narrow behavior.

2. Individuals resulting from crossover operations quickly die.
3. Most generated adaptors do not manage concurrency, but rather manage it to keep the involved components alive. It becomes clear that this is due to the non correlation of the two fitness measures (a fitness measure at client side and a fitness measure at server side).
4. If we modified the fitness measure, to include correlation of the different fitness values, we observed that other solutions were found. However, it was very difficult to create a suitable fitness function that guides the algorithm to an 'adaptor' solution. This problem has to do with the fact that measuring the working of an adaptor is very difficult and very conflict dependent.
5. Some adaptors implemented a feedback behavior that resulted in livelocks. From this experiment, we have observed that the liveness requirement is essential.
6. Comments with respect to figure 11.5:
   (a) With this actor we introduced a delayed reward. Only when a `SetPosition` could be executed to move to the right or the left, a reward is assigned. This requires the search algorithm to test out a number of different tracks until finally a reward can be assigned.
   (b) This adaptor works with two layers, hence the 4 transitions and different situations that can be encountered.

With respect to the before mentioned experiments two things became clear. First: we have found a suitable representation to express adaptors, however this representation does not easily produce concurrency solvers. Second, concurrency problems can barely be measured, therefore we started investigating how we can modularize the adaptor. To this end, we have split the behavior of the adaptor in three separate modules, as described in chapter 7.

11.7 Experiment 4: Bypassing a Provided Concurrency Strategy

Experiment: Automatic deduction of reachability analysis
Means: prolog conversion from Petri-net description and prolog analysis
Deployment: off-line
Input: Petri-net description in prolog and a required marking.
Output: A transition to execute to come closer to this marking.
Description: The description of this experiment is entirely contained in chapter 8.
Results: We have tested the working of this adaptor with relative markings and with absolute markings. To obtain absolute markings, we ran a number of test-components which produced an absolute marking from a real-life situation, afterward the prolog analyzer needed to find out how to enable that specific situation. In *all* cases the results were almost immediately known.

11.8 Experiment 5: Liveness and the Bucket Brigade

Experiment: Module to keep a client-component alive
Means: Classifier systems, the BBA and the earlier developed Petri-net Representation
Deployment: on-line
Input: Petri-net description of an interface, check-pointed component
Output: New transitions that will compete to keep the client component alive
Results:

1. The BBA is problematic because it specifies the mean future reward instead of a maximum future reward

2. One of the problems the BBA has, is that newly added transitions need an initial strength. If this strength is too low, the transition will be removed in the next step, without it having the ability to prove itself. If the strength is too high, slowly, good working transitions are flushed from the system. This can happen in situation that a new transition is never verified. To solve this problem we had to add a fairly difficult aging system, in which the strength of classifiers was determined based on their age.

After introducing the aging system to allow new rules to prove themselves, the similarities with reinforcement learning became clear.

11.9 Experiment 6: The Liveness Module and a Petri-net Representation

Experiment: Module to keep a client component alive
Means: Reinforcement learning, together with the earlier presented Petri-net representation
Deployment: on-line
Input: Petri-net of the interface, check-pointed component
Output: New transitions that will compete to keep the client component alive
Description: In the experiment, we use a number of different components, with different checkpoints, that will assign rewards as certain checkpoints are met (see section 9.2, page 146). We will not document where these checkpoints are set due to space considerations.
Results:

1. Numerical results and a discussion of them have already been given in section 9.4.1.
2. Works very well in all cases, but one. Delayed rewards form no problem at all.
3. If the reward function has hidden correlations which highly influence the reward, thus the problem being not a Markov Decision Process, the learner has much more difficulties finding out what is exactly required.

The representation we use in this experiment, together with the reinforcement learning approach seems suitable for the problem we are dealing with. In all tests a satisfactory keep-alive program would develop quickly. In comparison to the classifier representation it is easy to swap two arguments or to increase the value of an argument. In our Petri-net representation this is easy because a) we have variables which add an extra level of structuring (the algorithm will find one solution for all squares on the whiteboard instead of one solution / square) and b) the operations at our disposal are of a sufficiently high level to be useful.

11.10 Implementation

The implementation of all these experiments, together with the results are available online. We didn't put them in appendices because this would require around 400 extra pages.

- ⇝ `*.component` files need to be compiled to .java files by using the component system compiler.
- ⇝ `*.jjt` files need to be compiled with jjtree
- ⇝ `*.jj` files need to be compiled with javacc
- ⇝ `*.java` files need to be compiled with javac (watch the CLASSPATH environment variable)
- ⇝ `*.l2net` files are colored Petri-nets described in text. They should be converted to low level nets by the Petri-net compiler.

- ⇝ `*.l1net` files can be used as input in the Petri-net evaluator
- ⇝ `*.ini` files are input to the component system controller

11.10.1 The Component System

Runtime; Location: `http://borg.rave.org/cgi-bin/seescoacvs/component/system/*`
Runtime; Files: `Component.component, ComponentSystem.component, AbstractPort.java, AbstractMultiport.java, AbstractPortCreator.java, ComponentImpl.java, ComponentImplFactory.java, Connection.java, DirectPort.java, ExecutionLoop.java, Global.java, InterruptHandler.java, InvocationMessageHandler.java, Message.java, MessageDispatcher.java, MessageHandler.java, MessageHandlerCreator.java, MessageQueue.java, MultiPort.java, Port.java, SameHandlerCreator.java, Scheduler.java, SetupPort.java, SinglePort.java, StandardScheduler.java, State.java, StreamAble.java, StupidScheduler.java, SymbolTable.java, ThreadedInterruptHandler.java`
Compiler; Location: `http://borg.rave.org/cgi-bin/seescoacvs/component/parser/*`
Compiler; Files: `Transformer.jjt, SimpleNode.java`, and a large amount of parser node definitions. These start with `Ast.*`.
Controller; Location: `http://borg.rave.org/cgi-bin/seescoacvs/testcases/scss/*`
Controller; Files: `Controller.component, ConnectionProxy.component, ContractGenerator.component, ReceivingMonitor.component, SendingMonitor.component`

11.10.2 The Petri-net Evaluator Code

Location: `http://borg.rave.org/cgi-bin/seescoacvs/component/concurrency/*`
Evaluator: `PetriNetEvaluator.java, PetriNode.java, StaticNet.java, DynamicNet.java, ExecMessage.java, ExecutionTrigger.java`
Places: `Place.java, SinkPlace.java, SourcePlace.java`
Transitions: `Transition.java, InTransition.java, ArcEntry.java, InputEntry.java, MessageTransition.java, OutTransition.java, OutputEntry.java`
Expressions: `AgAddOperator.java, AgAndOperator.java, AgCombinedExpression.java, AgEqOperator.java, AgExpression.java, AgField.java, AgForeach.java, AgGlueOperator.java, AgGrOperator.java, AgIdentifierDescription.java, AgIdxOperator.java, AgInteger.java, AgMessage.java, AgMulOperator.java, AgNotOperator.java, AgOperator.java, AgRange.java, AgSet.java, AgSubOperator.java, AgToken.java, AgVariable.java, Applicable.java, Bindings.java, EqualsExpression.java`
Parser & Convertors: `PetriNetParser.jj, Printer.java, PetriNet2Prolog.java, PetriNetConvertor.java, PetriNetSimplifier.java, PetriNetNormalizer.java`

11.10.3 The Adaptor Code

Location: `http://borg.rave.org/cgi-bin/seescoacvs/component/concurrency/*`
Using classifiers: `Adaptor.component, ClassifierCondition.java, ClassifierList.java, ClassifierMessage.java, ClassifierMessageList.java, ClassifierRule.java, ConcurrencyCase1.java, ConcurrencyCase2.java, Gene.java, MessageArgument.java, Scenario2_1.java, Scenario2_2.java`
Component link: `MessageDescription.  java, MessagePlace.java, MyInvocationMessageHandler.java, MyInvocationHandler.java`
The Pittsburgh approach: `NeglectSynchro.component, PetriNetGenerator.java, PetriNetAdaptor.component, PetriNetLoader.component, PetriNetCrossover.java, PetriNetTimer.java, PetriNetCombiner.java, StandardLoader.component`
The liveness module: `FitnessAction.component, LearningAdaptor.component, ReinforcedAdaptor.component, GewogenPlace.java, GewogenTransition.java, PtReceiver.java`

The enforce action module: `anal.pl, ForceAction.component, ForceAction1.component, ForceAction2.component, PetriNetBackTracker.java`
The concurrency module: `SynchroAdaptor.component`

11.10.4 The Actor Components

Location: `http://borg.rave.org/cgi-bin/seescoacvs/component/concurrency/*`
Shared Classes: `ConcurrencyException.java, ActorRoot.component, Pos.java, ThreadCatcher.java`
Server Components: `PlayFieldView.java, PlayfieldCore.component, Square.java`
Client Components: `Actor*.component, BallActor*.component, FloodActor*.component, LineActor*.component, Actor*.l2net`

11.11 Summary

IN THIS CHAPTER we have reported on the experiments we have performed. Our solution has changed over time from an initial scheme-representation of a concurrency adaptor, to a modularized reinforcement learning Petri-net adaptor. The path we have followed is a standard, 'prototype, experiment, observe, adjust' cycle in which mistakes are corrected in later experiments.

The experiments have allowed us to understand that creating a learning algorithm that will immediately create a concurrency adaptor is not possible, because the no-race requirement is very difficult to express and to validate.

We also learned that an off-line strategy offers certain advantages to verify the quality of the representation. We tried four different representations of an adaptor before a suitable one was found. These are, a scheme representation, a single message reactive classifier system, a multi message reactive classifier system and a Petri-net representation. Because the latter suited our needs, we stopped investigating which representation should be used.

However, afterward we turned our attention to the off-line/on-line problem. All the experiments initially conducted used an off-line strategy. However, from a pragmatic point of view, it is very unlikely that an off-line strategy can be made to work in an on-line environment. The main reason for this is that, once an off-line strategy becomes fixed in an on-line environment it might be unable to learn what to do in new situations, which have never been present in the off-line context. Therefore, we turned our attention to an *on-line* learning strategy, under the form of reinforcement learning. This turned out to be a very satisfactory approach.

Chapter 12

Discussion

IN THIS CHAPTER we validate our approach by observing the experimental results. First, we will briefly summarize our approach and the results. Second, we make some important observations based on the experiments and third we present some technical limitations of the presented work.

12.1 Introduction

WE WILL NOW BRIEFLY summarize our approach and the experimental results. In our approach we created a general concurrency adaptor that can be used to mediate the differences between conflicting concurrency strategies: one server and multiple clients. As stated in the introduction section 1.3.5 on page 9, the adaptor we wanted to create should be able to mediate concurrency conflicts in such a way that no conflict arises. We have defined a conflict as a situation in which the mutually agreed behavior between the different communicating partners cannot be executed over the interconnection between them. In our case study, the whiteboard, the mutually agreed behavior between the different actors was not to cross each others boundaries.

The adaptor itself contains three modules. The first module is an enforce-action module which bypasses the concurrency behavior of a server component. This module makes use of a prolog program. The second modules is the liveness module. For every client component present, a liveness module will be created. Such a liveness module bypasses the required concurrency strategy in such a way that the client component can continue with its core functionality. The liveness module has been implemented by means of a reinforcement learning algorithm. Experimentally we observe that this approach is good *if* the client component returns correct rewards (that is rewards that are statistically correlated to the state of the involved Petri-net). The third module is the concurrency module, which takes care that no race conditions occur within the message order between the participating clients and the server. This module has been explained in chapter 10.

12.2 Observations about the Concurrency Adaptors

IN TABLE 12.2 WE give an overview of the discussed conflicts from chapter 6. The “description” column describes which conflict we are covering. The “possibly deadlock” column contains ’Yes’ if the approach used by the clients involved in the conflict could possibly lead to a deadlock. (for instance, a waiting locking strategy). The “Liveness” column specifies whether the liveness module works on the involved client. If a ’No choice’ is mentioned, the liveness module has not much choice to keep a component alive. For instance, a waiting locking strategy, where a `lock` request can only result in a `lockTrue`. ’Non Markov’ in this column means that the rewards are specified in such a way that the Markov property does not hold. The concurrency column specifies how the concurrency adaptor works. ’+Dead’ means that the concurrency module could possible deadlock, *if* the involved components exhibit such a behavior that might lead to a dead-

Conflict		Adaptor			Result
Description	Possibly Deadlock	Module			
		Liveness	Concurrency	Enforce	
6.1: Simple Syntax		Ok	Ok	Ok	Ok
6.2: Parameter Encoding		Ok	Ok	Ok	Ok
6.3: Simple Reentrancy	Yes	No Choice	+Dead	Ok	Ok
6.4: Async Reentrancy		Ok	Ok	Ok	Ok
6.5: Simple Control flow		Ok	Ok	Ok	Ok
6.6: Idle Control flow		Non Markov	N/A	Ok	Fail
6.7: Waiting client, Non-waiting server	Yes	No Choice	+Dead	Ok	Ok
6.8: Nested line server, square client		Ok	Ok	Ok	Ok
6.9: Non-nested line server, square client		Ok	Ok	Ok	Ok
6.10: Square server, field client		Ok	Ok	Ok	Ok
6.11: Layered server, non-layered client	Yes	Ok	+Dead	Ok	Ok
6.12: Layered client, non-layered server		Ok	-Dead	Ok	Fail
6.13:Transition		Ok	State	State	Ok
6.14: Nested Transition		Ok	State	State	Ok
6.15: Layered Transition		Ok	State	State	Ok
6.16: Transactional Client		Ok	State	State	Ok
6.17: Empty server	Yes	No Choice	+Dead	Ok	Ok
6.18: Upscale granularity	Yes	Ok	+Dead	Ok	Ok
6.19: Downscale granularity		Ok	Ok	Ok	Ok
6.20: Waiting clients, non-waiting server	Yes	Ok	Ok	Ok	Ok
6.21: Multi-multi conflict		Ok	Distri	Ok	Out Of Scope
6.22: Multi-multi conflict 2		Ok	Distri	Ok	Out Of Scope

Table 12.2: Overview of the conflicts and the adaptors.

lock. ’-Dead’ means that the concurrency module will possibly introduce deadlocks, even when the client components will not lead to a deadlock. ’State’ means that not only the resources need to be considered, but also the content of the resources. The “Enforce” column specifies how the enforce-action module performs on the server involved in the conflict. The result column states whether the overall behavior will be correct or not.

We will now observe some of the behavior of the 22 adaptors:

1. The idle control flow conflict (6.6) cannot be mediated because the rewards are non Markov, therefore it is impossible to get the liveness module to work.

2. The layered control flow conflict 6.12, occurs because a client expects a field-lock to be available, while the server doesn’t offer one. This conflict can be mediated. However, if multiple similar clients join, the result might deadlock, while the clients are specifically designed to avoid deadlocks by first obtaining a server lock. The reason why this layered conflict cannot be mediated is because the resource (the server-lock) is a virtual resource.This is discussed in more detail in section 10.6.6.

3. The transitional conflicts (6.13, 6.14, 6.15 and 6.16) can be mediated *if* the enforce-action module and the concurrency module take into account the state of the Petri-net places that are common to the involved components.

4. The two multi-multi conflicts (6.21 and 6.22) cannot be mediated because the concurrency module is not designed to be able to handle multiple servers.

Given the required mutually agreed behavior between the different actors: not to cross each other’s boundaries, we observe that in most cases (18 of the 20 tested conflicts) the concurrency strategy is mediated correctly and allows the components to execute as they intended. Below we will continue with some further observations we can make with respect to these results.

12.3 Concurrency Strategies are Very Simple

AN OBSERVATION WE CAN MAKE about the current concurrency strategies is that they are essentially very simple. Even the rollback-able concurrency strategies, which are not necessarily easy to implement, are often nothing more than finite state machines. This statement is based on two observations. First, our reachability analysis can deduce very quickly how to bring a server component in a certain state. Secondly, our liveness module learns very quickly what to do in a certain situation. This is to be expected to a certain extent because we have carefully created the representation of our situation-recognition system. However, this representation itself is also nothing more than a finite state machine.

If we think about interfaces (not only concurrency interfaces), it is not commonplace to hide features and functionalities within the API. On the contrary, most interfaces are designed to make it *easy* to access all the necessary operations and modify the state of an object/component.

From these two observations one might think that it would have been more appropriate to use finite state machines to describe concurrency strategies. This is not the case because the concurrency strategies themselves often contain parallel resources that act independently. This is very difficult to express with standard finite state machines. In section 3.7 on page 58 we have given two examples that demonstrates the need and advantages of using Petri-nets.

12.4 Technical Observations

IN THIS SECTION WE DISCUSS some technical limitations and observations we can make, based on the experiments we have performed.

12.4.1 Non-Incremental Approach: The Problem of Late Joiners

A limitation to our work is the problem of late joiners. We did not investigate what will happen when a new component joins a set of already communicating components. Currently, this situation is not allowed. However we feel this will not constitute an obstacle. A keep-alive module can be created for every joining component and the concurrency strategy can be extended to allow new components to join.

12.4.2 The Component System

During our research we implemented a new component system. We conclude from our experiments that this component system offers a lot of advantages, especially with respect to glueing components together. Its high level reified message representation, together with its disciplined non blocking messaging system allows for easy adaptor creation. In comparison with object oriented languages, where the problem of glueing together different libraries and programs is well known, this is an advantage. For instance,

- ⇝ The component system allows for writing adaptors in an easy, understandable way as discussed in chapter 2. Compared to object oriented languages, the problem of late binding, polymorphism an inheritance, which complicates the writing of adaptors, is not present.
- ⇝ The component system allows for transparent distribution because the model itself doesn't follow 'call' semantics and is connection based.

In appendix A we discuss how difficult it would have been if we would have used something like Java RMI to do this work.

12.4.3 Thread based versus Event Based systems

During the experiments we have used an event based component system. This allowed us to write adaptors easily and is a model representation of an open distributed system. However, open distributed systems are not the only area where conflicting interfaces occur and not all applications are written by using an event based approach. If we look at applications that make use of object oriented frameworks then we encounter similar problems because not always all upgrades of part of the application are entirely backward compatible [LSMD96]. A small alteration in the provided behavior or a wrong estimate of the usage scenarios can result in a complete application breakdown. This also happens for applications that dynamically link to external libraries (.DLL's offering extra functionality or plugins). Typical for this kind of object oriented application is its seemingly 'sequential' behavior over object boundaries. In fact, this kind of application is often written within a thread based context and they rely heavily on a shared memory (in the broadest sense of the word). However, the use of a shared memory as a communication medium between threads (or between different parts of an application) makes it very difficult to isolate the communicating partners and modify the flow of information between them. Since it is necessary to modify the information flow to resolve any conflicts it might be very difficult to use a similar approach as the one we have presented. Appendix A contains an extensive explanation of the difficulties of using such a thread based model.

12.4.4 Learning Algorithms

Another observation that can be made is that a randomized testing procedure (such as is implemented by different learning algorithms) is the best guarantee to write well-tested and robust software. Actually, our experiments took longer than expected due to the fact that our learning algorithm always created unanticipated situations. This happened in more than one area as discussed below:

- ⇝ *Petri-net evaluator*: We are now able to present a Petri-net evaluator which has been thoroughly tested, both the evaluation of all kinds of random expressions as well as the evaluation of randomized Petri-nets has been tested.
- ⇝ *Implementation of Actors*: On the actor level a lot of unexpected interactions were detected, such as in the flood fill algorithm 25 on page 147. During all our tests we were focused on a flood fill that slowly increases its terrain. However we specified a Petri-net that simply declares a non-nested locking strategy. This means that every lock can return a `lockFalse`. The checkpoint (1c) in the algorithm doesn't expect a possible `lockFalse` and will fail if a `lockFalse` is returned in the line before. This was only found out after measuring the fitness of the adaptor.

This has led us to use genetic algorithms not as a means to solve the problem of conflicting interfaces, but rather as a tool which helps in creating a suitable representation and which has helps in black box testing our Petri-net evaluator.

12.5 Observations about Petri-net Interface Descriptions

THE USE OF PETRI-NETS AS AN INTERFACE DESCRIPTION LANGUAGE allows for a better software development process. Not only is the code documented in a readable and formal way, the formal properties of the documentation allow for automatic testing of the underlying component. Given the Petri-net and a correct component we then can

- ⇝ check *completeness*: A testing program can take random walks through the Petri-net and check whether the underlying component works as expected: it does not crash, it does not end up in an invalid state and so on. When using timed Petri-nets, the timings can be verified and when needed memory-requirements can be verified against executed transitions.
- ⇝ check *consistency*: A testing program can verify whether all incoming and outgoing messages from and to the component are consistent with the Petri-net. For instance, it can check that all messages sent out by the component are enabled within the Petri-net.
- ⇝ *track* behavior of external components: By inserting a tracing component (section 3.6.3 on page 55) it is possible to verify whether external components honor the messages they are allowed to sent.
- ⇝ If we even go a step further, the Petri-net can help in *guarding* access to the component, by disallowing unexpected messages.
- ⇝ do a *formal analysis* to guarantee that no dead ends can be reached within the component. It is also possible to verify whether there is a home-state that is always reachable from any possible reachable state. This is important because this guarantees that a component can be reset at any time.

It is clear that the advantages offered by using a Petri-net description of an interface can outweigh the difficulties of writing one.

12.6 Performance

ALTHOUGH THIS DISSERTATION MAKES NO claim about performance, we will briefly discuss the performance of the entire adaptor. To do so we will discuss three separate estimates. For the entire adaptor we will investigate

1. $t_{adaptor}$: how much time is spend in handling one message ?

2. $m_{adaptor}$: how much memory is required by the adaptor ? We will express this in function of the minimum memory requirements of every involved component.

3. $n_{adaptor}$: how many messages are sent for every single message from a client to a server and vice versa.

To make these estimates we will investigate the timing and memory behavior of the three different modules. Afterwards we will investigate the message count. To be able to make these estimates we will need a number of variables:

- The number of server messages sent will be denoted n_{server}. A message is a server message if it is sent out by the server towards one of the clients.
- The number of client messages sent will be denoted n_{client}. A message is a client message if it is sent out by a client towards the server. The n_{client} refers to the sum of *all* messages transferred from any clients to the server.
- $\#clients$ which is the number of clients involved.
- In the upcoming discussion we will sometimes need an estimate of how long it takes to verify which transitions are enabled in the Petri-net. Theoretically (as explained in section 3.4.4 on page 50) this is exponential because one might need to verify all combinations of tokens versus free variables. However, in practice such situations seldom arise. As we have observed experimentally, the verification whether a transition is enabled can be done in $O(1)$. However, in the upcoming discussion we will consider $t_{transition}$ to be the mean time *over all Petri-nets* to verify whether a transition is enabled. Afterward, we will, depending on whether we want a worst case estimate or an expected estimate, fill in $t_{transition}$ with an appropriate expression.
- We also need a description of the memory-requirements of a Petri-net. We will denote this size m_{server} or m_{client}, depending on which Petri-net is involved. Intuitively the size required to represent a Petri-net is at most the size needed by the underlying component itself. For instance, in our case the server Petri-net describing the state of the whiteboard (of size x by y) will need $O(x \times y)$ tokens. The server component itself also needs to keep track of the same information. Petri-nets describing *more* different states than recognized by the involved components are less useful (see Petri-net guidelines, section 3.8 on page 63).

We will now start with discussing the time- and speed- estimates of the liveness module, the enforce-action module and the concurrency module.

12.6.1 The Liveness Module

Giving a performance estimate of the liveness module is difficult because the performance of this module largely depends on how well trained it is. Initially it might need a large number of messages and rewards to understand which behavior exactly is required by the client component. However, once a correct behavior is installed it will behave more or less constantly, depending on how much exploration is allowed. Here, we will only consider the case where the liveness module exploits correctly learned information.

To give an estimate of the time spent within the liveness module, we observe that the liveness module needs to find out a) which transitions are enabled and b) select from the enabled transitions an appropriate one by means of Q-learning. Step a) can be done in $t_{transition}$ by using the locality property of Petri-nets. Step b) can be done in $O(1)$ because the number of active 'added' transitions is kept constant. During execution of the liveness module, non-fit transitions are removed and new transitions are added. The process of removing a transition is $O(1)$ and the process of adding a new transition is also $O(1)$. This results in a time estimate of $t_{liveness} = O(t_{transition})$.

The memory requirements of the liveness module are the memory requirements of the Petri-net describing the state of the client component and the newly added transitions. Because these newly added transition are bounded to a certain maximum, we consider it to be a constant. The size of the Petri-net is bounded by m_{client}. Hence $m_{liveness} = O(m_{client})$.

Later on, when giving a performance estimate of the entire adaptor it should be taken into account that the concurrency adaptor contains multiple liveness modules. Therefore we will add an extra index to the liveness measure: $m_{liveness_i} = O(m_{client_i})$ and $t_{liveness_i} = O(t_{transition})$

12.6.2 The Enforce Action Module

Giving a performance estimate of the enforce action module is even more difficult. We currently rely on a prolog implementation to decide how to enable a certain action. This makes it difficult to give correct estimations. Potentially both the memory estimate as well as the timing estimate might be exponential in function of the necessary search depth, $O(c_1^{searchdepth})$. So the worst case time estimate is $t_{enforcer} = O(c_1^{searchdepth})$. The worst case memory estimate is $m_{enforcer} = O(m_{server} + c_2^{searchdepth})$ because we also need to take into account the memory-requirements of the server side Petri-net.

However, as observed during the experiments the search depth is always very small and most concurrency strategies offer the possibility to reach a certain state quickly. Therefore we will consider a practical time estimate of $t_{enforcer} = O(1)$. And, similarly, a practical memory estimate of $m_{enforcer} = O(m_{server})$.

12.6.3 The Concurrency Module

The concurrency module needs to keep track of all the enabled transitions for every client component indirectly connected to it. For every client i a set of controlled transitions and managed transitions will be kept in memory. The Petri-net of the server is not necessary within this module. This gives a memory estimate of $m_{concurrency} = \sum_{i=1}^{\#clients} O(2.m_{client_i})$. The time spent within the concurrency module is a constant. For every incoming request a cross-table is checked and a decision is taken immediately. This leads to $t_{concurrency} = O(1)$.

12.6.4 Message Sends

To determine how many messages are sent between the client components and the server component we will consider two cases: functional messages and synchronization messages.

Functional (Logic) Messages

Every functional message going from a client toward a server will go through several connections:

- the "client component $\leftrightarrow$ liveness module" connection
- the "liveness module $\leftrightarrow$ concurrency module" connection
- the "concurrency module $\leftrightarrow$ enforce action module" connection
- the "enforce action module $\leftrightarrow$ server component" connection

Messages coming from the server and going to the client will pass the same stages in reverse order. This means that for functional messages the number of messages equals $4.n_{client} + 4.n_{server}$. These results can be slightly improved by merging the concurrency module and the action-enforce module into one module at the server side. Hence, we could have an estimate of $O(3.n_{server} + 3.n_{client})$.

Synchronization Messages

The behavior of synchronization messages is not as simple as the behavior of logic messages. The problem comes from the fact that synchronization messages are not simply passed through from module to module, but are a) altered to represent the possible choices between the liveness module and the concurrency module and b) are not present between the concurrency module and the enforce action module. However, this observation does not prohibit us from making a global observation about the synchronization communication: We know that the synchronization messages introduced by the adaptor is minimal to support all components: the enforce action module will only send out the necessary synchronization messages, while the liveness module will have learned which synchronization messages are favored by the client. From this observation we know that the adaptor will not introduce useless synchronization messages. This already allows us to give an estimate of $O(n_{server} + n_{client})$. However, to estimate how many extra messages might be added we give a trace of a synchronization request from client to server:

- $\leadsto$ client component $\leftrightarrow$ liveness module: the client component sends a synchronization message to the liveness module: n_{client}.
- $\leadsto$ the liveness module $\leftrightarrow$ concurrency module: communicates the changing state and possible choices to the concurrency module: n_{client}.
- $\leadsto$ concurrency module $\leftrightarrow$ liveness module: in the worst case scenario, the concurrency module must always make a choice and communicate it back to the liveness module: n_{client}. In this performance estimate we will assume that this will only occur when the real server would in practice also return a message, hence: n_{server}.
- $\leadsto$ the liveness module $\leftrightarrow$ client component: the liveness module sends back a synchronization message to the client component: n_{server}.

The communication of messages from the server to the client happens only when a certain functional message arrives at the action enforcer for which synchronization messages are required. Such a message trace looks like:

- $\leadsto$ enforce-action module $\leftrightarrow$ server component: when necessary a message will be sent to bring the server in the required state. So, we can assume that this is correlated with the number of client synchronization messages: n_{client}.
- $\leadsto$ server component $\leftrightarrow$ enforce-action module: this is captured in the n_{server} variable.

This results in a number of messages of $3.n_{server} + 3.n_{client}$

From the results of the functional messages and the synchronization messages we can conclude with an estimate of $O(3.n_{server} + 3.n_{client})$.

12.6.5 Overall Performance

From the previous performance estimate we can now summarize the worst case performance:

$$n_{adaptor} = O([3.]n_{client} + [3.]n_{server})$$

$$t_{adaptor} = O(c_1^{searchdepth}) + \#clients.O(t_{transition}) = O(c_1^{searchdepth} + t_{transition})$$

$$m_{adaptor} = O(c_2^{searchdepth} + m_{server}) + \sum_{i=1}^{\#clients} O([4.]m_{client_i})$$

If we consider, as explained before, that the time to verify a transition's state (enabled vs disabled) is constant and we assume that the prolog implementation is able to deduce a correct path in $O(1)$ then we can give an expected estimate of:

$$t_{adaptor} = O(1)$$

$$m_{adaptor} = O(m_{server}) + \sum_{i=1}^{\#clients} O([4.]m_{client_i})$$

This means that we expect that the adaptor needs as much memory as required by the different states of *all* involved components. Also, the time necessary to handle one message is constant.

12.7 Motivation vs Solution

IN THIS SECTION WE WILL DISCUSS our initial motivations and compare them with the solution we have presented. Our main focus here will be to investigate how our prototype could be made to work within the context of open distributed systems, multi agent systems and embedded systems.

12.7.1 Open Distributed Systems

Our adaptor as it is now does not run automatically in open distributed systems. Below we discuss two obstacles still remaining. The first being adaptor deployment (or who decides to run an adaptor) and the second being the problem of meta-conflicts (conflicts between the involved Petri-net descriptions).

Adaptor Deployment

A very simple fact about our adaptor is that, before it will be used, *somebody* has to insert it somewhere. We will now discuss the implications of this.

Typically, the client is the first one to observe any conflicts with the server. This means that the client will be the first to *want* to introduce an adaptor. This is not necessarily easy because

- ⇝ To insert an adaptor, extra formal documentation, describing the behavior of the client component, is needed. If the software used on the client does not provide such a documentation then it might be difficult to run an adaptor.
- ⇝ Our adaptor needs the possibility to isolate the server from its environment. Hence, the client side cannot solely take the decision to insert the adaptor because it requires cooperation of the server side.

The server side on the other hand is often the last one to observe any conflicts and might not want to introduce an adaptor for a number of reasons:

- ⇝ The server administrator needs to make the decision to insert an adaptor. Technically speaking this doesn't form a problem. However, (s)he might feel the clients should upgrade.
- ⇝ The performance penalty of the adaptor is considered to be too high.
- ⇝ Inserting an adaptor might requires a quality tracking process (how well does it work, are there any performance penalties, does it introduce bugs, is the end-result more stable than the initial situation and so on) which might take too long a time.
- ⇝ The extra formal documentation, describing the behavior of the server, might not be available.

⇝ The required extra formal documentation might not be offered by the clients. In such a situation the server needs to 'know' the correct definitions of different clients. This results in conflicts because the server-side interface descriptions can be incompatible with the actual clients.

Some of these problems are relatively subjective and closely linked with the involved business model, however two major observations still remain. Firstly, not everybody might agree to follow our standard of Petri-nets as a formal documentation technique. Secondly, the formal documentation itself might lead to conflicts. We will discuss these two problems below.

Why would somebody want to use Petri-nets ?

One of the motivations behind this research was the fact that standards in open distributed systems are sometimes merely used as a means to stay in control of how software is used instead of being a means to communicate with other software. If the intention of a company is to be incompatible to force partners to make certain decisions, then there is nothing that will prohibit such a company from doing so, not even Petri-nets. However, the description of this behavior was only used as a motivational example and not as the problem we wanted to solve.

On the other hand, companies that want to be compatible with other software will find that our proposal of using Petri-nets offers a big advantage that renders their usage worthwhile. The strength of the documentation format comes from automatically checking completeness and consistency. Completeness being that all messages that a component understands and uses are described within the Petri-net. Consistency being that the documentation is compatible with the behavior of the component. Both properties can be checked automatically. In other words, Petri-nets help the developer in the software development process by offering better and more accurate documentation.

Meta-Conflicts

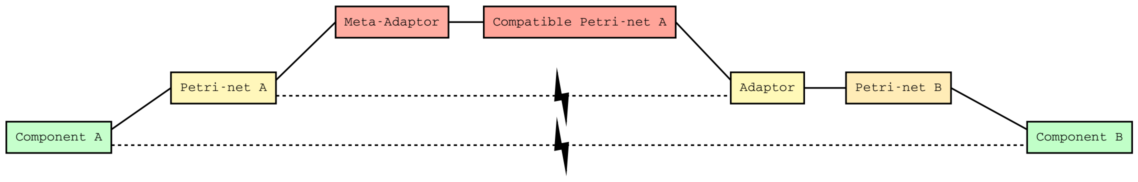


Figure 12.1: Stack of adaptors

In section 1.1.1 on page 2 we argued that defining one standard in open distributed systems is unrealistic because competitors will always try to adopt the standard and unexpected incompatibilities will arise. This problem could also occur between the different formal interface descriptions. This kind of conflicts are meta-conflicts. The problem of meta-conflicts is wide ranged; from slight discrepancies between the Petri-net descriptions to conflicts with entirely new, possible better, formal descriptions.

If the formal interface description are entirely compatible then it might be possible to create a super interface description that can express all details of the different formal descriptions. The only thing that needs to be done then is implementing converters from the different interface description languages to this super interface description language. However, it could also be possible that required information in one interface description language is simply not present in another interface description language. This would complicate this method. However before this kind of meta-conflicts can be investigated, a number of realistic conflicts between interface description languages need to be investigated. Only then can this problem be approached pragmatically. This remains future work.

If there are only slight differences between the Petri-net format used, or between the terms used within the Petri-net then other approaches can help:

- ⇝ *meta-adaptors based on Petri-net evaluation.* Such a meta-adaptor is pictured in figure 12.1. If we would create an adaptor similar to the one we have created for conflicting concurrency strategies then we need a 'common ground' that can be used to verify the behavior of the different Petri-nets. In our case, the common ground was the compatible functional behavior. In the case of conflicting Petri-nets a common ground could be the fact that a Petri-net can be verified for completeness and consistency. By tracing the incompatible net and recreating a compatible net during such a trace it should be relatively easy to convert an incompatible net to a compatible net. This can be investigated in more detail in future work.
- ⇝ *automatic deduction* of the states and transitions within a component by inspecting the code and data segments in a typical communication. If we could avoid the need for Petri-nets of an interface description at all, then the problem of meta-conflicts might be avoided. How the interface of a component can be learned is discussed in section 13.8.5.

12.7.2 Mobile Multi Agent System

A second motivation for doing this work was the problem of concurrency management in mobile multi agent systems. Because communication happens in a peer to peer manner between agents, we now discusses how our solution relates to the problems of peer to peer systems. First, we discuss how our adaptor preserves the required behavior if no central locking server is used. Second, if a central locking server is used to coordinate the behavior of a group then we see how our approach fails to offer a solution.

The Adaptor Preserves Behavior

In this section we illustrate how our adaptor correctly mediates concurrency problems in a peer to peer application that makes no use of a central coordination server.

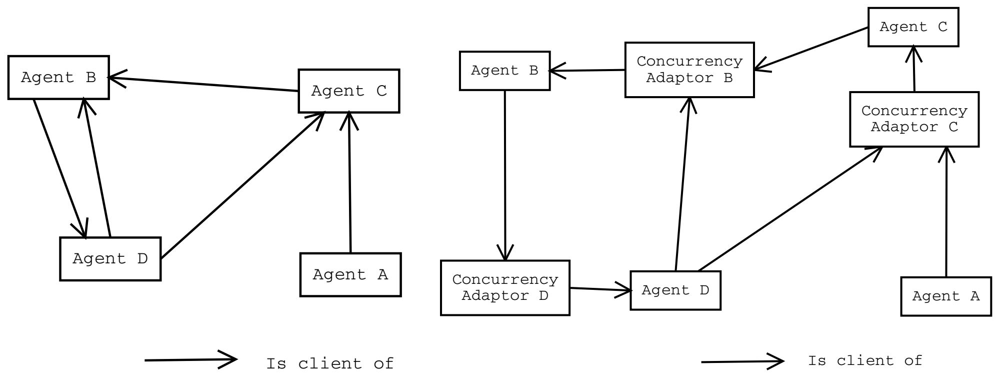


Figure 12.2: Peer to peer communication between agents in a mobile multi agent system. The arrows indicate which agents require behavior from another agent. The left figure is the normal interconnection without inserting adaptors. The right figure is the same interconnection but with adaptors inserted at the server ports/interfaces.

In a mobile multi agent system there is at first sight not really a difference between client-components and server-components. However, if we look at the behavior they require or provide (or in other words, which agents implement a certain functionality), then we can actually make the difference between a client component and a server component. A client agent will expect a certain functionality from another agent, while a server agent will provide a certain functionality. However, contrary to a typical client-server setting, in this situation every agent can be a server as well as a client. To easily support such a model we will assume that the server behavior of such an agent is offered on one port, while the client behavior is offered on another port. This allows us to introduce adaptors on every agent (peer) that offers a service. The resulting interconnections are pictured in figure 12.2.

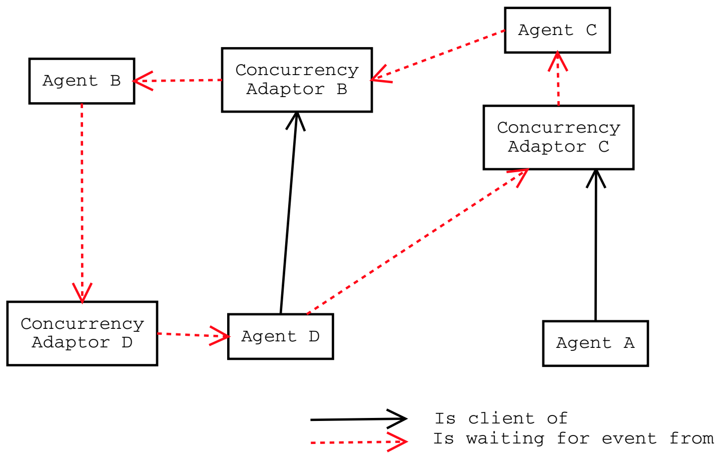


Figure 12.3: Deadlock over the different adaptors in a peer to peer situation.

Because every agent can be a client as well as a server, it becomes difficult to identify a 'session'. In the typical client-server architecture, the client manages a session and tries to get things done on the server. Within a peer to peer system, sessions hop from one agent to another and cannot be localized anymore in one agent. This might lead to deadlocks between communicating peers, without the possibility for the adaptors to do something about it. This is illustrated in figure 12.3. The adaptors cannot resolve such a deadlock because they are part of the waiting loop. However, this problem has little to do with the ability of the adaptor to 'mediate' the conflicting behavior. In fact, if the application is programmed in such a way that it relies on concurrency strategies that lead to unwanted behavior then this unwanted behavior will also occur with adaptors, otherwise it will not.

The Adaptor does not Preserve Coordination

In a peer to peer application there can also situations where our adaptor fails to work. The example we present to illustrate this uses a central locking server that is used by all components to identify sessions and control access to the application's resources.

In peer to peer systems it is often necessary to bring multiple peers from one consistent state to another consistent state. This however requires from every peer the ability to obtain a number of locks, distributed over different peers. Once all the locks are held the session can apply the changes on the involved resources and unlock them again. Often this is done by passing session id's from one peer to another such that receiving peers can verify whether the resources they hold can be accessed by the incoming session. However, the session id's themselves are often provided by other components than the receiving peer itself. For instance, consider a transaction server such as pictured in figure 12.4. The transaction server will be used by any peer that needs a transaction id (this will be our session id). This transaction id will then be passed from peer to peer (without going through the transaction component).

In this setup, multiple concurrency strategies can be present. First there are the concurrency strategies offered by one peer towards another peer, secondly there are the concurrency strategies offered from one peer toward the transaction server and thirdly there is the concurrency strategy offered from the transaction server toward the peers. We will assume that we have mediated the conflicts by placing at every server interface a concurrency adaptor. This might lead to problems:

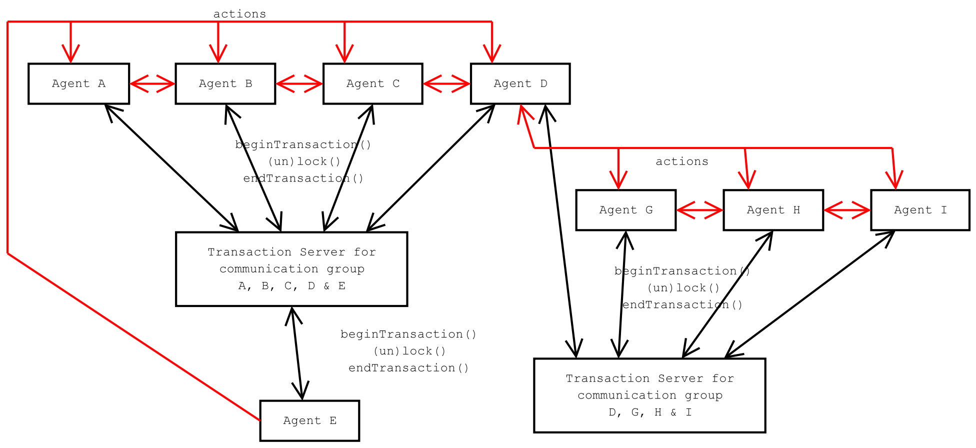


Figure 12.4: Distributed Transaction server.

⇝ Our adaptor requires the availability of a compatible functional behavior. This poses a problem because the only functionality our central transaction component offers is about synchronization. If the transaction component is hardwired with respect to the resources then we can use the common set of resources to be the functional link between them. However, if the transaction component uses *first class resources*: i.e. resources that are published by clients and recognized by the central server, then it might be very difficult to find any useful common behavior.

⇝ If peers that have no notion of session id's are pulled into the application then our adaptor will not provide them with this notion. This means that their resources will be freely available to everybody wanting to lock them, effectively destroying the use of the central transaction server. So, if we allow components to join in an 'open' way then this kind of conflict cannot be mediated.

⇝ Even if peers have a notion of session id's then we still have a problem because the application is using first class session id's and the information about such a session is localized at one central place, while in fact all the concurrency adaptors on the peers need to know about these session id's. E.g: if agent E requests a session id from the transaction server, then the Petri-net in the liveness module connected to agent E, will recognize the session id to be owned by agent E. However, if this session id is passed to agent A, then this agent will be completely lost, because the Petri-net in the liveness module of agent A, connected to agent E does not recognize this session id.

⇝ If we would be able to resolve these problems, (by not using first class session id's or first class resources) another problem might occur: synchronization races due to *hidden communication*. Because our adaptor might delay the passing through of a synchronization request until really necessary, a resource might not yet be locked if the agent wants to access it through another channel. Even worse, this resource might never become locked if the agent doesn't access it directly on the transaction component.

From these examples we see that a) first class resources, b) first class sessions, c) hidden communication and d) the possibility that the concurrency strategies simply cannot be mediated, still form a major obstacle. Investigating how these problems can be solved remains future work.

12.7.3 Embedded Systems

One of the motivations behind this dissertation were conflicts between connected embedded systems. The rise of smaller and mobile embedded devices increases the chance of encountering interface conflicts. In this section we investigate how suitable our solution would be for this kind of domain. First we must note that the biggest difference between desktop software and embedded software is the constraints put on it. Among others, there can be memory-constraints, speed constraints, bandwidth/latency constraints and real time constraints. Secondly, aside from these constraints, there is also an entire process involved in creating embedded systems. Typically first a prototype is developed which is, as the project continues, scaled down into a much smaller artifact. During the entire process, quality control is of utmost importance, because, especially for consumer devices, it is not always possible to fix a bug once the device has been sold [SEE99]. From a high level point of view this provides an opportunity for our adaptor because it decreases the likelihood of bugs introduced due to concurrency strategy conflicts. However, as embedded systems are developed pragmatically, we will now discuss how it would be possible to reduce the requirements of our adaptor in such a way that it could fit within certain constraints.

Memory constraints

Very often embedded systems are limited in their memory. There are two approaches that might help in reducing the memory requirements of the adaptor:

1. Given the performance estimation earlier

   $$m_{adaptor} = O(m_{server}) + \sum_{i=1}^{\#clients} O(4[.]m_{client_i})$$

   we see that the adaptor requires 4 times as much memory as for every Petri-net describing the behavior of a client. For desktop software this doesn't pose much problems, however for embedded systems it would be nice to be able to reduce this number. Collapsing the different modules into one might help, not because it reduces the amount of required 'information', but because the overhead of storing this information can be reduced. For instance it is possible to store the information on whether a resource is controlled and/or managed into the transitions of the Petri-nets themselves.

2. A second important place where storage can be optimized is our prolog module. The prolog interpreter is needed to determine how we can force the server to be in a certain state. To resolve this problem we can either try to remove it or try to make better use of the interpreter.

   (a) *Removing the prolog interpreter*: this could be done by integrating the adaptor more tightly with the server side component. By removing the concurrency strategy at the server component we can make progress on two ends. Firstly, removing the server side concurrency strategy reduces memory-requirements with respect to data space (to store locking information) and code space (to implement the concurrency strategy). Secondly, *not* implementing a concurrency strategy at the server component removes the need for an enforce-action module.

   (b) *Better usage*: If it is not possible to remove the concurrency strategy at the server (because the embedded system is not under control of the one inserting the adaptor) then we can make better use of the interpreter by implementing the other modules (the liveness and concurrency module) also as prolog programs. Making use of an interpreter in itself would also drastically reduce the memory-requirements and might even be better in comparison to compiling the adaptor. However, this could also introduce a speed penalty. So, it is the standard tradeoff between speed vs place requirements.

Bandwidth Constraints

Certain embedded systems have low bandwidth constraints, which means that they can only send and/or receive a small amount of bytes per second. As stated in section 12.6.4, our current implementation of the concurrency adaptor requires 3 times more messages than there are messages posted by the client and server component together. This could weigh heavily in a low-bandwidth environment. Therefore we now investigate how we could reduce the bandwidth used by the adaptor by decreasing the amount of extra message sends between the modules or by making the penalty of doing so less severe.

1. *reducing the amount of extra messages* can be done straightforwardly by combining all modules in one embedded system such that there is no extra communication necessary between the modules. It should also be noted that by decreasing the number of messages that need to be sent, we might not only optimize bandwidth, but also the latency of client/server communication and the internal speed of the adaptor (as explained in the next section).

2. *decreasing the penalty*: if it a) is possible to modify the server such that it doesn't offer any concurrency strategy or b) we can place the enforce-action module on the server-side component, then it is possible to *decrease* communication towards the server because no *synchronization* messages ought to be posted between the concurrency module and the server-side. The only remaining messages would be functional (logic) messages.

Speed constraints

There are two distinct approaches one can use to make the adaptor more efficient with respect to speed:

1. One of the biggest bottlenecks in the adaptor is its modularized nature. This gives rise to sending three times as many messages as there are messages being communicated. By demodularizing the adaptor and allowing the modules to work together in one shared data-space it is possible to increase speed substantially. Instead of copying the messages, pointers to the messages could be passed. Actually, by doing so it could be possible to have no overhead at all in 'extra' communication'.

2. As described in section 3.4.4 on page 50, Petri-nets can be implemented in hardware very efficiently (data flow machines). This could make it possible to reduce the time necessary to verify whether a transition is enabled, $t_{transition}$, with a certain factor. It should be noted that such a hardware implementation will only provide a marginal improvement and not *solve* the combinatorial problem of searching for a matching set of tokens. Still, for embedded systems, a speed increase with a certain factor is always welcome.

3. Another possibility to increase the speed of an adaptor is by parallelizing the process. By implementing every module within a separate piece of silicon it is possible to increase speed linearly.

Real time constraints

Real time constraints often require a much better quality control process [SEE99]. This process should be extended to our adaptor as well.

$\leadsto$ Embedded systems requiring real time constraints should not rely on 'learning' approaches, because these might behave indeterministically. Nevertheless, if such a system would be build anyway, then the deployment phase should be carefully investigated. The learning algorithm should, during an off-line training-phase, be allowed to learn but when placed in an on-line setting it should no longer learn further.

⇝ Systems constrained by hard runtime deadlines should not rely on synchronization operations that might block the system for an unknown amount of time. Depending on the behavior of various clients it might be possible that the concurrency adaptor decides to set certain, possible important, real time events in wait. In case this should be unavoidable a formal analysis of the entire software within the embedded system should be made. This should of course be done *after* the system has been scaled down from a prototype to an optimized artifact.

In this section we have discussed how our adaptor, as it is, could be used within embedded systems, how it can be scaled down to fit extra non functional requirements.

Chapter 13

Conclusions

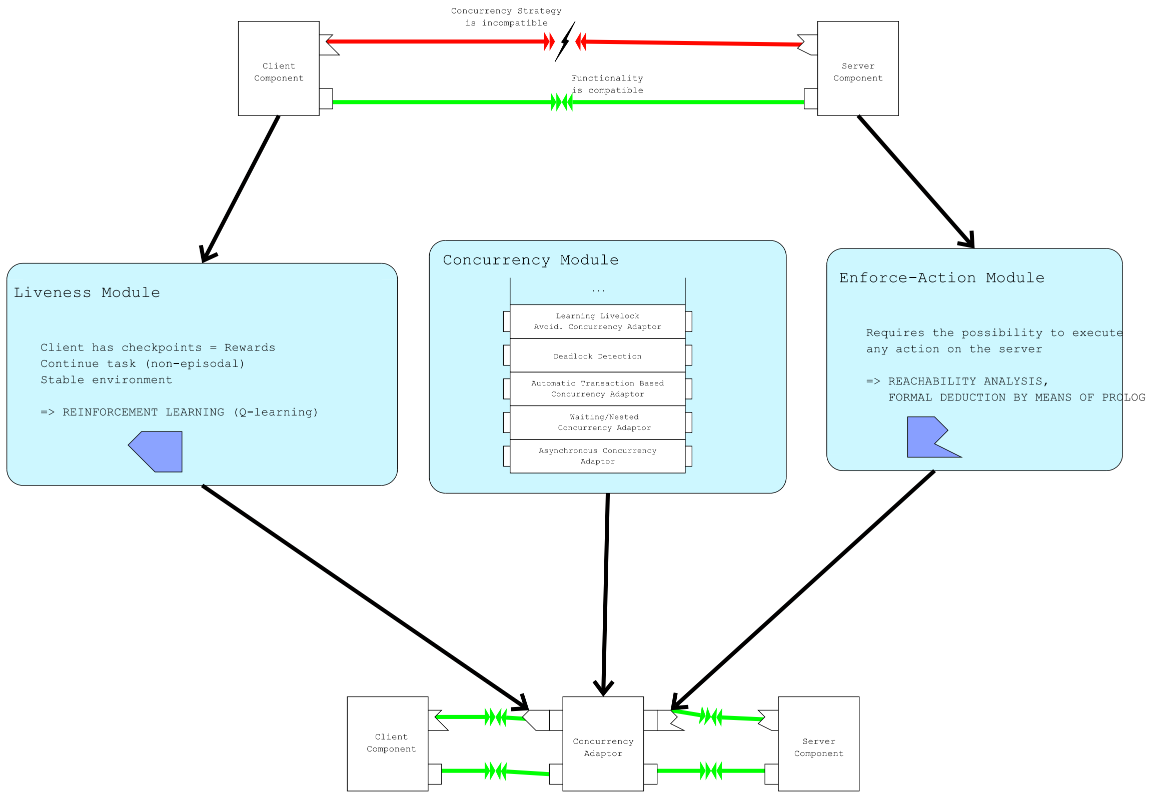

Concurrency Strategy is incompatible
Client Component
Server Component
Functionality is compatible
Liveness Module
Client has checkpoints = Rewards
Continue task (non-episodal)
Stable environment
=> REINFORCEMENT LEARNING (Q-learning)
Concurrency Module
...
Learning Livelock Avoid. Concurrency Adaptor
Deadlock Detection
Automatic Transaction Based Concurrency Adaptor
Waiting/Nested Concurrency Adaptor
Asynchronous Concurrency Adaptor
Enforce-Action Module
Requires the possibility to execute any action on the server
=> REACHABILITY ANALYSIS, FORMAL DEDUCTION BY MEANS OF PROLOG
Client Component
Concurrency Adaptor
Server Component

Figure 13.1: High level module overview

IN THIS CHAPTER we first summarize our work. Then we will elaborate on our scientific contributions as well as on the applicability of the thesis. We point out the limitations of our work, give guidelines how an adaptor for other conflict domains might be created and finish the dissertation by pointing out possible future work.

13.1 Technical Summary

BASED ON THE CASE STUDY of conflicting concurrency strategies, we have shown how intelligent adaptors can be created automatically. We have illustrated this by creating such an intelligent adaptor as a sequence of three modules, where every module has different responsibilities.

The liveness module learns how to keep the components behind the concurrency strategy alive. For the concurrency adaptor as a whole, such a module is necessary to offer to the underlying components an optimal feedback behavior, as opposed to for example, returning a behavior that leads to endless live-locks such as `lock-lockfalse-lock-lockfalse` ad infinitum. Because it is still unknown whether liveness is a decidable property of Petri-nets, we have decided to use a reinforcement learning algorithm (Q-learning and an ϵ-greedy strategy) to approach the problem. Feedback for the learning algorithm comes from the underlying component by means of statically placed checkpoints. Every time a checkpoint is reached a reward is sent to the adaptor and the learner can, when necessary, modify its future behavior. The reinforcement learning algorithm itself has been very efficiently mapped onto the structure of colored Petri-nets, in such a way that storage requirements are minimal, without losing essential information. This optimization is mainly based on a) the explicit state information of the Petri-net already present and b) the creation of a system that can recognize certain situations and knows how to handle them. Our situation-recognition system is expressed as a Petri-net. During execution, new transitions are constantly added to the Petri-net. The learning algorithm automatically explores the new avenues offered by these transitions and, when suitable, these new transitions are kept alive and the new situation is remembered. The reason why we use Petri-net transitions as situation handlers has been explained thoroughly, and possible alternative representations have been investigated. To verify the suitability of our representation we used genetic algorithms.

The concurrency module of our concurrency adaptor inserts an appropriate concurrency strategy. To do so, it communicates with the two other modules about resources (for instance the possible squares in a whiteboard) and actions (for instance a `setPosition`). There is no explicit annotation of resources and actions on resources, both are implicitly represented within the Petri-net. This information is automatically deduced by the first module under the form of enabled transitions and equivalent states. This, together with extra information of the liveness module that describes which futures are possible and which futures the component would like, enables this module to make correct decisions about what to do. Once such a decision is taken the client component can perform its actions on the server component.

The enforce-action module bypasses the concurrency strategy of a component entirely. This allows for more freedom in the two other modules. The enforce-action module does this by offering a logic interface to the outside world while inserting the necessary synchronization messages at the appropriate times. To do so, the module needs to know how certain markings within the underlying Petri-net can be reached. In this dissertation we have achieved this by means of a reachability analysis in prolog.

In these three modules, the adaptor needs a) to analyze how to enable certain actions, b) to know how to reach a certain state and c) to know which resources are involved. In order to be able to know this the adaptor requires a formal documentation in the form of Petri-nets. Every component needs to provide a Petri-net describing its own required or provided concurrency strategy in a consistent and complete way. Consistent means that any action that is allowed by the Petri-net should be accepted by the component and complete means that no action that is allowed upon the component is undocumented.

13.2 Thesis Statement

WE STATED that in open distributed systems, one needs intelligent adaptors to mediate interface conflicts. We identified three reasons why one might need an intelligent adaptor: a) in open systems, it is very unlikely that one standard will be used by everybody, b) the highly volatile nature of peer to peer computational networks gives rise to interface conflicts and c) with the current advent of interconnected embedded systems and ambient intelligence the problem of conflicting interfaces will become even more prominent. We claimed that *it is possible to create, for certain categories of concurrency interface conflicts, a concurrency-adaptor that automatically learns how to resolve the conflict in such a way that a) the required behavior of the involved components can be executed over the adapted interconnection and b) it is able to work on-line in certain open system.* We have validated this by constructing such a concurrency-adaptor. Below we discuss the different requirements and how we have satisfied them.

To show the necessity for such adaptors, we have introduced our real-world case of conflicting concurrency strategies. We have investigated different concurrency strategies and explained why components in open distributed systems *need* a concurrency interface. Relying on this analysis we discussed the problem: not everybody will write the same concurrency interface for his components and thus concurrency interface conflicts will arise. We identified and explored in much detail a large number of practical conflicts. More specifically, we investigated conflicts that can easily occur when one or both of the concurrency interfaces is upgraded without prior notification and without full backward compatibility.

To show that we can create such an intelligent adaptor we have constructively developed one. The adaptor itself needs detailed information of the provided and required interfaces of all participating components. The adaptor works by means of three modules. It contains two modules that essentially bypass the provided or required interface and one module that actually implements a suitable concurrency strategy. To bypass a concurrency strategy we investigated two techniques. The first technique was a reachability analysis of a Petri-net, the second technique was a reinforcement learning algorithm. Finally we explain how resources and actions can be detected automatically within a Petri-net. This chain of three modules constitutes an intelligent adaptor that is able to set up a link between conflicting concurrency interfaces, for which the adaptor does not know the exact interface beforehand. The adaptor is restricted to client server architectures. In section 12.7.2 we have explained in detail what can go wrong with peer to peer applications. Essentially, the adaptor does not work in open peer to peer systems because not all communication between different parts of the application can be mediated by one adaptor.

13.2.1 Abstract Requirements

We now reconsider the abstract requirements from the introductory section.

The adaptor should be able to work on-line

This requirement was necessary to make the adaptor useful in an open system. The performance estimates as well as the setup of the adaptor enables it to work on-line. The liveness module is an on-line prolog program that can deduce how a certain state can be enforced upon the server, without needing to 'test' the server's behavior. The concurrency module is a program that can be executed on-line and the liveness module has been specifically designed, by selecting Q-learning as a learning technique, to work on-line.

The adaptor should work by *only* modifying the message flow between the involved components.

The adaptor is a central component that is placed solely on the connection between different components. Hence the only modifications it can introduce is on the messages between the different components. Therefore, this requirement has been satisfied.

The adaptor should resolve conflicts

The most important abstract requirement of an intelligent adaptor is based on the notion of an *overall required behavior.* In our case study the overall required behavior between the different conflicting concurrency strategies was to avoid actors crossing each others boundaries. As we have experimentally observed, our adaptor supports this implicit requirement. Two cases can occur:

1. all the components agree on an overall behavior and offer/require a concurrency strategy.

2. some components do not offer a concurrency strategy

In the cases where there is a full agreement, we have experimentally observed that our adaptor works. This has been described in chapter 12. If there is no full agreement on the required behavior then the adaptor should try as hard as possible to satisfy as much as possible of the required behavior. This requirement is satisfied because our concurrency module allows the presence of an actor that does not manage certain resources. In such a case these actors will be set 'on hold' until a suitable moment arises to realize the required changes.

13.3 Contributions

THE MAIN CONTRIBUTION OF THIS THESIS is the process we have been following to create an adaptor. Instead of trying out one technique and verifying whether it works or not, we have kept on searching to find a (set of) suitable technique(s). To reduce the complexity of the problem we have restricted ourselves to the case of concurrency conflicts, which we have expressed in an event based model. When necessary we have invented new techniques (The extremely useful Petri-net documentation for describing interfaces and a new technique to verify the bias of computer generated programs), or combined existing techniques (Petri-nets & Reinforcement learning, the three different modules that constitute an adaptor). This has lead to the creation of a concurrency adaptor that mediates differences between conflicting concurrency strategies. Below we summarize the contributions.

1. *Open systems need concurrency interfaces:* We showed that the components of an open distributed application *need* concurrency interfaces. The reason is that one cannot foresee how a component in an open distributed system will be used, therefore one cannot know which actions are supposed to be atomic, and therefore every component should allow other components to specify a critical section. Hence the components themselves need to offer a concurrency strategy. We have illustrated this extensively by means of a whiteboard.

2. *Open systems require concurrency adaptors:* We have shown that components within an open distributed application can offer all kinds of concurrency strategies and that it is very likely that concurrency interface conflicts will arise between different components. We have extensively discussed a set of real-world concurrency interface conflicts and we used these conflicts to validate our claim.

3. *Petri-nets as a formalisms for interface description*: We showed how Petri-nets can be used to describe the behavior of an interface in a formal way. This formal interface description is useful because it can always be synchronized with the source-code, as consistency and completeness can be checked in an automatic way.

4. *A concurrency adaptor:* We showed how an intelligent protocol adaptor can be constructed in order to resolve incompatibilities between communicating components. Such an approach is indispensable to cope with the combinatorial explosion of protocol adaptors that are needed in an open distributed setting, when dealing with interconnected embedded devices or when writing components. In all these cases components interact with other components in unpredictable ways.

5. *Automatic bias verification:* We showed how genetic algorithms can be used to verify a representation instead of providing a solution. We consider that if a certain problem can be solved easily under a certain representation, it is a good representation. If the same genetic algorithm has more trouble finding a solution the representation might not be so suitable.

6. *Unification of Petri-nets and reinforcement learning:* We showed how reinforcement learning can be unified with colored Petri-nets in an efficient way by exploiting the structural properties of the colored nets.

7. *Component based software engineering allows for easy adaptor creation:* We showed how an event based model allows for easy adaptor creation if it is used in a disciplined way, that is, if it is purely event based and no compromises are made to integrate with other models, such as a thread based models, and that no communication happens outside the system.

8. *Guidelines for developing interface adaptors:* This dissertation can be used as a platform that offers recommendations and guidelines for the development of conflict adaptors for other domains.

13.4 Limitations

IN THIS SECTION WE ELABORATE ON SOME LIMITATIONS OF THE PRESENTED APPROACH:

1. *Only conflicting concurrency strategies*: Conflicting interfaces represent a major problem that has to be dealt with. To approach this problem we have investigated an accessible but smaller problem of concurrency conflicts. This limits the immediate applicability of our technique. Nevertheless,

   (a) the case itself is chosen such that its correct working *cannot* be measured easily at runtime. E.g; How can one measure a race-condition ? This clearly depends on what the components consider to be a valid state. By using such an interesting requirement, we have investigated an interface conflict that cannot be automatically deduced from a state-machine (such as done in [Wyd01]). Also, another requirement of our concurrency conflicts was the fact that the adaptor should keep the components alive. This also made it difficult to use one standard technique to solve the problem. So, our case is indeed a small subset of all possible conflicting interfaces, nevertheless it is certainly not a trivial case.

   (b) in the domain of open distributed systems this work is an important contribution because the problem of conflicting interfaces is often overlooked and because our work allows for a decoupling of the required and provided concurrency interfaces. If we would want to extend this research to other application domains, then a substantial part of it should be redone. For instance, it might no longer be feasible to use Petri-nets, or to use a liveness module and/or enforce action module. Later on we will discuss a case in which conflicting socket libraries are investigated.

2. *Well-working functional link:* An important limitation of our work is the fact that we rely on the availability of an existing, well-working link between the conflicting components. More specifically, we can only generate an adaptor on non-functional requirements between two interfaces if the core functionality of both components is compatible. Such an assumption is not unexpected because it helps in creating a predictable environment that allows for verification of the generated adaptor. E.g, a whiteboard offering `set_position` (with underscores), while a client expect `SetPosition` (without underscores) are clearly in conflict. Because this is a conflict on the *logic* interface between both agents, our adaptor will not be able to mediate such a conflict. Some approaches might help to solve this kind of conflicts

   (a) Instead of relying on a correct functional link, we could probably also rely on another reference frame for which the correct working of the adaptor could be measured. For instance, a requirement such as 'make sure that both partners draw a moving actor on the whiteboard' could result in a guiding principle for an off-line learning algorithm. Later on, we will give an example of conflicting socket libraries to illustrate this.

   (b) Similarly, other research requires the availability of a placeholder that describes the required interaction [Wyd01]. This research however is very limited to the requirements it can describe.

   (c) Depending on the application it might be sufficient to just rely on the commonalities between the expected and provided *state* of the involved components instead of relying on the common *actions*.

3. *A Controlled Case:* We have only been using one case, a whiteboard, to illustrate concurrency strategy conflicts. We did not investigate other examples. Nevertheless, this whiteboard case has been explicitly created to illustrate practical problems. With it, we were able to illustrate race-conditions, deadlocks, livelocks and transactions in such a way that the nature of open distributed systems was still preserved. From an academic point of view, this case describes everything one might need in open systems. From an industry point of view, a lot of problems have been avoided. For instance, in the 'real world' not everybody agrees to follow an event based model. Also not everybody will agree to write Petri-nets, some might even disagree with the fact that a concurrency strategy is necessary. However, instead of seeing such a controlled case as a weakness, we would rather like to see it as one of the strengths of this dissertation. By using a simple model, we were able to describe all necessary issues, from simple locks to transactions in such a way that a) their need was clearly demonstrated b) many of the concepts, necessary in the real world are present (resources, actions, processes,....) and c) the common problem, recognized by different solution providers are present (livelocks, deadlocks, transactions,...).

4. *Component based development:* We have limited our work to conflicts between components in a component based system. We did not investigate how our work can be unified with models that have a concept of shared memory. Neither did we investigate how our results can be mapped on thread based models. The main reason for doing so is that in an experimental model it is impossible to take into account all possible technical details. Appendix A covers a detailed explanation of what kind of problems we have avoided by doing so.

5. *Only one component system:* In this dissertation we have limited our research of adaptor creation to one component system, which we created ourselves. This was mainly due to practical reasons. The project that financed my research, the SEESCOA project, needed a definition of components and an implementation of a component system in Java. It was only out of practical considerations that we have used this component system because it allowed us to use it actively on more than one front. First, in the test-case for the SEESCOA project and secondly as a means to carry out our own research. By doing so, we were able to deliver a reusable and high quality component system. A second reason why we favored the SEESCOA component system is because it has a mature design. After creating components (agents) in the Borg mobile multi agent system, some design flaws came up, that were not easy to fix. These design flaws have been corrected in our current implementation of the component system. Aside from practical considerations, the component system itself is also representative for open distributed systems, applications within open distributed systems and can be considered as a common denominator of existing component systems.

6. *Technical limitations:* The adaptor created in this dissertation works when the involved conflicts arise from syntactical conflicts, the ordering of messages, re-entrance conflicts, conflicts between static resources and conflicts on the layering of concurrency strategies. However, this certainly does not cover all conflicts that might arise. For instance, our adaptor fails to work when:

   (a) *First class resources* and/or *sessions* are present and used as discussed in section 12.7.2.
   (b) There is *hidden communication* between peers. Because our adaptor does not see this communication it cannot adapt it. (section 12.7.2)
   (c) There are *non-sequence requirements.* These are requirements that have nothing to do with the sequencing of messages. Such requirements can be
      i. *timing*: For instance, a requirement such as 'every 2 seconds an alive notification should be sent'

ii. *memory*: For instance 'when handling this message, the memory should remain below ...'

iii. *state* information: For instance, "after a rollback the component should be in such a state". In our case this can be relatively easily added by using the places of the Petri-net.

13.5 Application Scope of our Adaptor

IN THIS SECTION WE discuss the application scope of our adaptor, first, without giving examples of such applications. We discuss why completely open systems are not feasible and why our adaptor is useful even in situations where exact requirements on the interface, but *also* on the usage of that interface, are necessary. Afterward we give some immediate practical applications of our adaptor.

13.5.1 Truly Open Systems: Feasible ?

Some conflicts are impossible to mediate. In section 6.1.2 on page 103 we have given examples of such conflicts. A notable example of such a conflict was our line example. In this example we assumed that the line actor would always have a lock on a square somewhere on the whiteboard. Another client required the possibility to lock the entire whiteboard. If those two actors were to work together on the same whiteboard then the second actor (which wants to lock the entire field), would never have a chance to do something because it would never be able to lock the entire whiteboard (simply because the line actor will always keep one square locked). This conflict is notable because it demonstrates that even realistic locking strategies can easily lead to impossible to solve concurrency conflicts. In this example the conflict arises from a subtle difference within the usage scenario involved.

This indicates that truly open systems, such as advertised by the mobile multi agent scene, might be impossible to build: a small change to a component or to one of the involved usage scenarios might transform a 'possible to mediate conflict' to an 'impossible to mediate conflict'. For open systems this means that probably the only way these systems can work is if *everybody* agrees on *exactly* the same *specification.* This specification should be exact (formal) with respect to what kind of usage scenarios and what kind of interfaces are allowed. This is clearly not realistic for *open* systems.

13.5.2 Then: Why Do We Need Adaptors ?

The statement 'open systems require an agreement between all partners to follow exactly the same specification' might sound contradictory with our initial motivations. However an exact specification does not prohibit from allowing certain degrees of freedom. An adaptor might shift the problem of being compatible with a very narrow interface description to being compatible with a wide range of allowed interfaces (within certain degrees of freedom). We explain this by elaborating on the different possibilities (all pictured in figure 13.2)

⇝ When the interfaces and the usage are *narrowly specified*, then we have an interface that is barely usable by other components [Ore98]. If somebody wants to use the component in a slightly different situation (a situation with other usage scenarios) it will no longer be compatible with the exact specification of the usage scenario. A component with such a strict requirement is very limited with respect to the client components it can serve.

⇝ When the interfaces are *narrowly* specified, but the component usage is *wider ranged*, then possible client components have, depending on their goals, more freedom in using the provided interface. However, this poses a problem because certain usage scenarios are not necessarily easy to implement over the narrow interface. E.g, implementing a field-lock

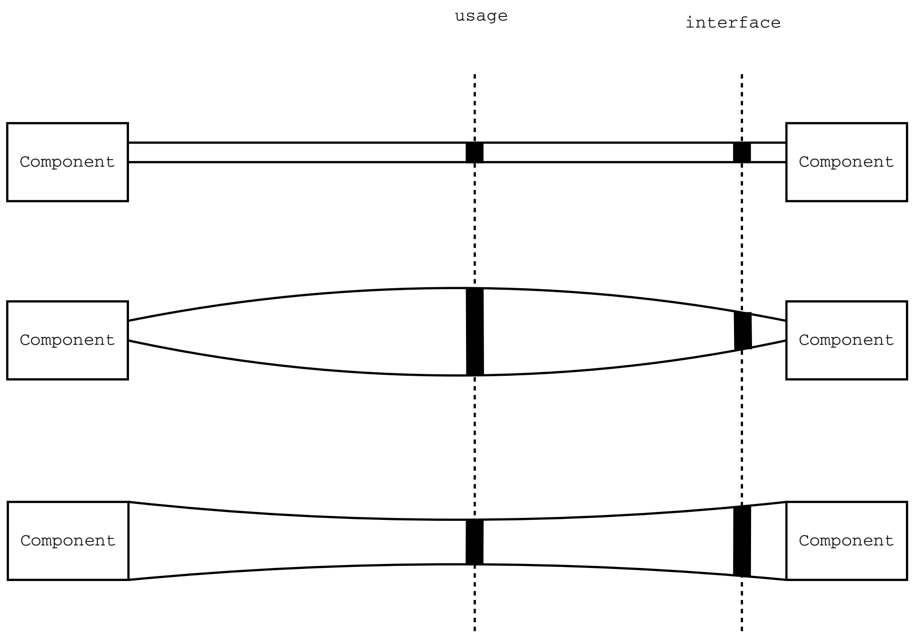


Figure 13.2: Interface boundaries and usage boundaries. Top of picture: small interface, limited usage. Middle: small interface, large usage range. Bottom: large interface, large usage range.

over an interface that offers the locking of squares. If we could, in such situations, allow other, slightly different, interfaces to access the same functionality then the development of open systems would become easier.

⇝ When interfaces are specified as wider ranged interface-descriptions (instead of exact syntactical descriptions) and the usage scenarios are wider ranged as well, then one component can use another as it sees fit. E.g.: one component specifies a reentrant locking strategy, while another specifies a non-reentrant locking strategy. However, before such wide ranged interface can communicate some conversion needs to take place. This is possible if we are able to mediate the differences between the different interfaces. This is where our adaptor comes in because it enables such scenarios.

To summarize, within certain boundaries, an automatic adaptor is useful because it allows any component a *choice* in the interface it provides or requires. This can have a major impact on the way components are developed as we will explain below. Also, it essentially offers a decoupling of the required interface from the provided interface.

13.5.3 Impact on Component Development

We initially started our investigation trying to foresee what the Internet would be in the future. We tried to foresee what kind of problems will arise. We concluded that conflicting concurrency strategies would become a large problem. In the approach we have presented we tackle this problem by assuming that every component will offer a concurrency strategy and that these will need to be mediated. In doing so, the problem of 'writing correct adaptors' is shifted to the problem of 'specifying the required/provided concurrency behavior precisely'. The result of this approach is that the developer does not have to implement the adaptors directly, but that he instead has the responsibility of writing a complete and consistent Petri-net. The developer can use such a Petri-net to write down a strategy specifically suited for the problem at hand. Instead of sticking to one of the standard concurrency approaches he can now easily specify

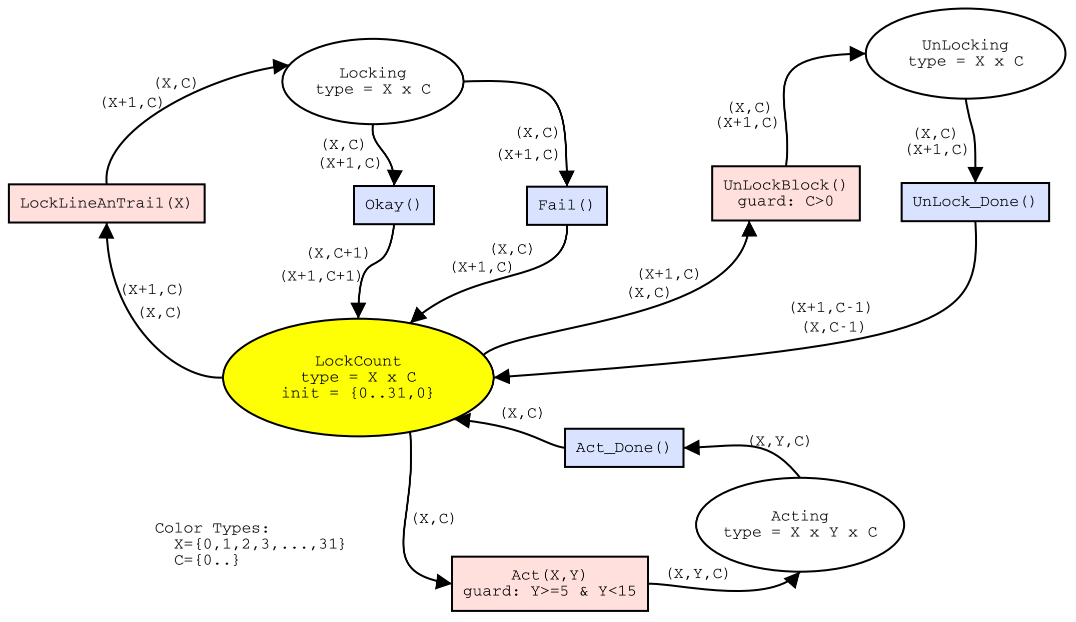


Figure 13.3: Demonstration how a Petri-net can be written that will immediately lock all fields necessary for the line and its trail.

which resources need to be locked simultaneously. Depending on the required usage scenario it might be possible to require (or provide) better suited interfaces for the needed functionality. For instance it is possible to write a client component that specifies that it wants to lock an entire line and its trail immediately. This is pictured in figure 13.3 and would make writing our line actor more easy. On top of this is the fact that this Petri-net exactly specifies what is needed and thus makes mediating conflicts involving this concurrency strategy even more easy.

13.5.4 Decoupling Client / Server

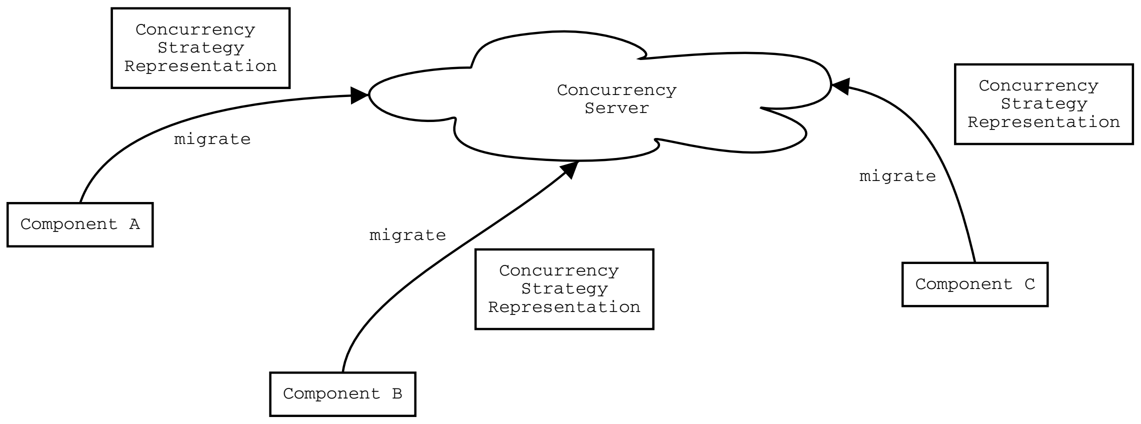

Concurrency Strategy Representation
Concurrency Server
Concurrency Strategy Representation
migrate
Component A
migrate
Concurrency Strategy Representation
migrate
Component C
Component B

Figure 13.4: Using a concurrency server to which components can send a concurrency representation agent.

If we follow this line of thought we end up with concurrency servers on which different components can publish the resources they have and the *behavior* they need. This might become even more interesting if, instead of supplying such a server with a Petri-net of the behavior, we could supply such a server with an actual program, that represents our original component. This program can express even more behavioral information than our Petri-net descriptions. In these cases mobility of components would be very useful. This is pictured in figure 13.4.

13.6 Some Practical Applications

Below, we describe some applications or application domains where our concurrency adaptor, as presented here, is useful. We first give a number of examples where the adaptor is useful in situations where the interface not necessarily changes. The applications we have been looking for were constrained by a number of requirements

- ⇝ The application should have a client-server architecture, otherwise our adaptor is not immediately applicable.
- ⇝ A concurrency strategy is necessary to guarantee a correct overall behavior
- ⇝ the resources should be statically defined.
- ⇝ when possible, the interface should change often, because this will naturally lead to numerous conflicts. However, this is not strictly necessary to make our adaptor useful.
- ⇝ the system must be event-based, or expressible within an event-based system.

Afterward we give an example of an interface with frequently changing semantics in which this research can provide an added value, without relying explicitly on the previous requirements.

13.6.1 Collaborative Computer Supported Work

A computer collaborative supported work environment often contacts a central server that offers some kind of whiteboard (in the broad sense of the word). Users can join such a whiteboard and through it communicate with each other, while still following some rules. In this case, the client program offers a user interface to the end-user, while the central server stores and manages the data. In such a multi-user environment it is necessary to lock operations on the whiteboard. Depending on the kind of locking strategy required by the client an adaptor might be necessary. This is a case where locking is necessary, the resource can be described statically, the clients *might* require a different concurrency strategy and it can be considered as an event based system. Therefore, our adaptor is useful in this context.

13.6.2 Databases Access

Databases are another area where our work might be useful. Databases are often centrally placed servers that allow different clients to update records. To guarantee that this happens in a consistent way, a locking interface needs to be present. If such a database would, together with the schema's of the different tables, also supply a Petri-net that describes the offered concurrency strategies, then it might be easier for clients to access this data because they in turn can define a more suitable required concurrency strategy. This can make implementing clients for databases much easier.

13.6.3 Frequently Changing interfaces: TCP/IP Sockets

We now present an example of conflicting interfaces within a domain other than concurrency interfaces: socket interfaces. Later on we will use this example as an illustration of how (part of) the techniques used in this dissertation can be used to implement adaptors in other domains.

When we were developing Borg, the biggest problem of creating a working version for all platforms was, contrary to what might be expected, not the user interface, but rather the socket

	Linux		Windows '98
Operation	2.2.*	2.4.*	
initialization	/		`WSADATA wsadata` `WSAStartup(1,&swadata)`
opening	`socket()`		`socket(...)`
binding	`bind(...)` hostname *might* append domain name		`bind(...)` obtaining local address very difficult
signaling of events	`signal(SIGIO,...)` `signal(SIGURG,...)`		polling necessary
connecting	`connect(...)` synchronous	asynchronous	`connect()`
send	error when packet too large		no error when packet to large
local circular sends	no deadlock	might deadlock	
recv	seldomly incomplete receives		almost always incomplete receives
closing	`shutdown(s,SHUT_RDWR)` `close(s)`		`closesocket(s)`
blocking	`i = fcntl(s,F_GETFL) & ~O_NONBLOCK` `fcntl(s,F_SETFL,i)` `fcntl(s,F_SETOWN,get_pid())`		`int i = 0` `ioctlsocket(s,FIONBIO,&i)`
nonblocking	`i = fcntl(s,F_GETFL) | O_ASYNC | O_NONBLOCK` `fcntl(s,F_SETFL,i)` `fcntl(s,F_SETOWN,get_pid())`		`int i = 1` `ioctlsocket(s,FIONBIO,&i)`
error constants	`EAGAIN` `EWOULDBLOCK` `ECONNREFUSED`		`WSAEWOULDBLOCK` `WSAEWOULDBLOCK` `WSACONNREFUSED`
transmission error	`errno`		`h_errno`
connection error	`errno`		`h_errno`
host error	`h_errno`		`h_errno`

Table 13.1: Some important, barely documented, differences between Linux sockets and windows sockets.

libraries. We have developed Borg for PalmOS, Windows, Linux and Macintosh. Every operating system offers its own version of a socket library. This has posed major problems because not every socket library is as easy to use as any other socket library. Table 13.1 contains the most important differences between Linux and windows.[1] Between the two presented API's some important differences exist:

- ⇝ *Syntactical* differences on multiple levels. The error constants under windows are all merged into `h_errno`, while under Linux, the error constants are separated in two variables, `h_errno` and `errno`. Another syntactical conflict can be found when marking a socket non-blocking.
- ⇝ Initialization of the library is different. Linux doesn't require initialization, Windows does.
- ⇝ Windows will not signal when data is received, hence the application needs either to poll constantly the sockets it manages, or it should start a new thread. Linux will signal when data might be available, hence not requiring a new thread or polling loop. This is a conflict that is not easy to solve in a platform independent way.
- ⇝ Between different Linux versions other conflicts might arise. In the kernel version 2.2 series, circular sends would not deadlock but return an error, while in the 2.4 series such a circular send will wait and deadlock. Depending on the usage scenario this might form a problem.

As illustrated by this example, interfaces that often get a new implementation, so called hotspots, will automatically give rise to differences between implementors. This often leads to conflicts. In this example the availability of Petri-nets to describe the semantics of the different functions would have been extremely helpful as well as the possibility to mediate the differences automatically. When we present the guidelines, we will investigate this example in more detail and give a sketch how to approach the problem of conflicting socket interfaces.

13.7 Guidelines

CONFLICTING INTERFACES REPRESENT A major problem that has to be dealt with. To approach this problem we have investigated a smaller, but more accessible problem. The solution presented in this dissertation is not a general solution in the sense that it can be straightforwardly applied to other domains. Nevertheless, in identifying solutions to a subset of a smaller problem, we have tried to keep both the solution and the subset of problems as general as possible. This allows us to identify guidelines which can help shaping new solutions to other problems of conflicting interfaces. We will now discuss these guidelines and explain them by giving an example of how the previous identified problem of conflicting socket libraries might be solved. This example will be presented in another font.

13.7.1 Explore the Environment

The first thing that needs to be done is to explore the environment. The best way to get grip on a certain domain is to explore existing real world conflicts and investigate what exactly causes the conflicts. In this dissertation we have done this by identifying a set of concurrency conflicts (chapter 6). The process of exploring the environment should result in a set of variabilities (that need not to be strictly orthogonal). In a later stage the variabilities are used as input for defining the requirements and possible solutions.

Once this is done, we should check whether it is necessary to represent these conflicts in a scaled down model. This might be necessary to reduce the technical problems involved and/or to speed up the process of testing conflicts (the prototyping cycle). In this dissertation we have

[1]The version for Macintosh and PalmOS have been omitted but can be found on the site `http://borg.rave.org/cgi-bin/borgcvs/borg/cborgcore/`. The different files are named `StdLinux.h`, `StdLinux.c`, `StdWindows.h`, `StdWindows.c`, `StdPalm.h`, `StdPalm.c`, `StdMacintosh.h` and `StdMacintosh.c`.

created our own component model instead of using standard CORBA services or other available techniques (chapter 2).

Important questions:

1. what kind of conflicts do we want to investigate ?
2. what are the variabilities ?
3. what kind of test conflicts will be investigated ?
4. what kind of prototyping model will be used ?

Example: Our socket case has already been described in section 13.6.3. We have identified a number of variabilities such as

- $\leadsto$ *blocking vs non-blocking calls*: a call can be either blocking or non-blocking. A non-blocking call returns immediately, while a blocking call will wait until the kernel has finished doing his job.
- $\leadsto$ *signaling vs polling* vs *waiting*: will the kernel inform us when something interesting (like an incoming packet) has occurred or do we need to poll/wait constantly?
- $\leadsto$ *syntactical* conflicts: how are the different calls named ? Are the same constants available, what is their meaning ?
- $\leadsto$ *initialization* of the libraries, *opening* of sockets and *shutting down*.
- $\leadsto$ *timing* of different calls. How long does it take before a timeout occurs ?

To identify a set of conflicts we can rely on a number of different applications ported to different platforms and kernels, such as we have done with the Borg mobile multi agent platform. In this case we don't need an actual model, however in defining the environment we can decide to start using a cross-compiler because this makes it easy to quickly test different programs.

13.7.2 Goal & Reference Frame

Once the environment is explored, it is time to look for a suitable reference frame. A reference frame is part of the environment and is common to all involved conflicts. The reference frame should be chosen very carefully because, later on, it should allow for a measurement of the correct working of an adaptor. In this dissertation, we have used as a reference frame the assumption that the 'functional link' between the involved components was compatible (chapter 7). If we would have chosen a reference frame such as 'every component can send a message', then we would not have been able to define what a good working adaptor was exactly. Therefore, when choosing a reference frame, it is important to find good answers for the following questions:

1. What is the goal of the adaptor ?
2. Where do you want to place the adaptor ?
3. What are the commonalities ?
4. Which commonalities are useful ?

Example: In our socket case, we try to make user applications, written for certain socket implementations, compatible with other socket implementations, without modifications to either the application or the kernel/library source code. We do this by inserting an adaptor between the application and the used library. Towards the application our adaptor will offer all the necessary socket primitives and from the underlying kernel/library, it will use the offered socket primitives. Our socket case has a number of commonalities over different implementations.

- $\leadsto$ Everything is about *communication* between to computers. Clearly our adaptor works if it is able to communicate with another program that makes use of any other socket library.

- ⇝ Everything is also about what kind of data is sent out over/received from, for instance, an Ethernet cable. If we could compare the data one socket library puts on the cable with the data another one wants to put on the cable then we have a common ground.

From these commonalities we will use the last one as reference frame because the protocol is the largest common denominator between different socket implementations and because it does not require a second computer to agree to the protocol you are using.

13.7.3 Identify Necessary Extra Information

Once a) the goal of the adaptor and b) a usable reference frame have been identified, it becomes necessary to explicitly state the requirements of our adaptor. Before we can state the requirements more formally, it is often necessary to introduce extra information. By analyzing the variabilities and the problems of expressing a requirement, it becomes clear what kind of extra information is needed. In a later stage we will need this information to create the requirements for the adaptor.

Some standard questions for *every variability*:

1. can we express it as a requirement ?
2. if so, how easy-to-check can we make this requirement ?
3. if not, what kind of information is missing ?
4. can this information be obtained *automatically* ?

Example: We go back to the variabilities we have identified earlier.

- ⇝ Clearly to be able to handle syntactical conflicts, blocking vs non-blocking conflicts, signaling vs waiting vs polling conflicts, we need the ability to alter the event flow between libraries. This is only possible if we at least *know* what kind of event flow is requested and/or provided.
- ⇝ A second important issue is *timing*. Socket libraries often rely on timeouts on all kinds of operations (sends, receives, connection acceptance and so on). To be able to react to such requirements it is necessary to understand them. Hence, the extra documentation should contain timing information.
- ⇝ Without the knowledge of when a certain block of data is put on the Ethernet cable, it is virtually impossible to understand the working of a socket library. Therefore, we need *communication* information.

Summarized, we need a) message flow information (captures blocking / non-blocking, signaling / polling / waiting and syntactical conflicts), b) protocol information (which data is put on the cable by which function) and c) timing information. None of these is directly available in the standard syntactical API accompanying socket libraries.

13.7.4 State the Adaptor Requirements Explicitly

Once the necessary, but missing, extra information is identified, we should be able to state the requirements explicitly. To do this we can again rely on the variabilities we have identified earlier. Which of these variabilities does the adaptor need to understand ? Can we define what a good working adaptor is for such a variability based on the reference frame ? In this case it might be necessary to re-investigate the reference frame, however we consider this to be part of the prototyping cycle. When defining the requirements it is important to distinguish between different kinds of requirements. These are:

1. *Some requirements can be statically checked* (off-line), without running the program, only by looking at the two conflicting components a program can be generated that will satisfy the requirement. This is a most favorable situation, however such a situation will not always occur. In this dissertation we didn't encounter such a requirement.

2. *Some requirements can only be checked by running the program.* For instance, a requirement such as the liveness-requirement cannot be automatically satisfied before program execution (off-line) because it requires runtime-rewards from the client component.

3. *Some requirements cannot practically be checked.* It should be noted that it is not because a requirement cannot be verified (or is too resource consuming to verify), that it is impossible to solve it. In this dissertation, the no-race requirement, is such a requirement. Checking whether no race-condition occurs at runtime is much more difficult than solving race conditions. It is clear that this kind of requirements is much more difficult to satisfy because it requires expert knowledge of the domain and knowledge of standard programming idioms to tackle these kind of problems.

Example: We go back to the variabilities we have identified earlier. We assume that the protocol that is used by the conflicting socket libraries is the same *and* compatible. We also assume that the information communicated by this protocol is compatible. So, the essential problem is in how the application communicates with the socket library to achieve a certain effect. With this we are able to specify the requirements informally:

- ⇝ The application should be able to use any event structure it requires to realize a data stream. The application will provide the adaptor with the required ordering of 'events'.
- ⇝ The application should be able to use any timing structure it requires. The application will provide the adaptor with these requirements
- ⇝ The adaptor should only use the provided socket implementation to realize the required behavior. The socket implementation provides a description of its event structure and its timing structure.

In the above, 'event' refers to a call, a return, a signal, a data block sent or received. This requirements are relatively informal, if this adaptor would be made, a more strict set of requirements would be necessary.

13.7.5 Correctly Represent Missing Information

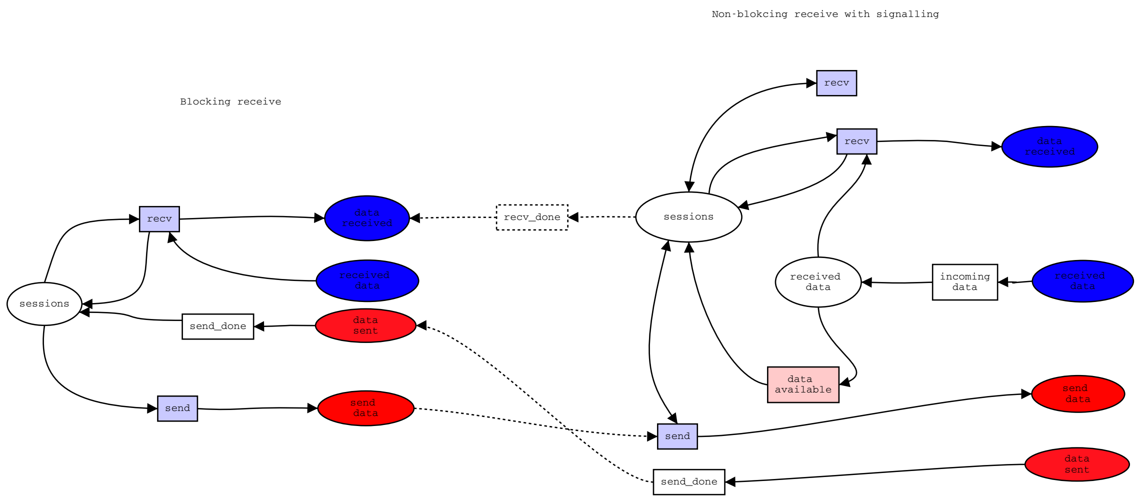


Figure 13.5: Petri-net description of a socket net.

The missing information must come from somewhere. Either the user of the adaptor supplies it, or the implementors of the different interfaces have to supply it. Essentially where it comes from does not matter, as long as it is there. To motivate users to supply the missing information, it might be helpful to make it useful. In this dissertation we have done this with our Petri-nets, they not only help in creating an adaptor, but they also help in detecting all kinds of interesting features of an interface/component (see chapter 3).

Secondly, the process of writing down the missing information should itself not be more difficult than the process of writing an adaptor. Therefore, it is necessary to carefully create a language that is able to capture all essential information in such a way that it is intuitive and expressive enough. However, here it should be avoided to introduce a Turing-complete language because then a number of the requirements might suddenly become impossible to verify. So, keep an eye on how the requirements can be validated afterward.

Questions with respect to the missing information are:

1. how can we make it useful to the one who needs to supply it ?
2. how can we present it in a language that is natural to humans and compact ?
3. how can we avoid a language that is Turing-complete ?

Example: For our socket case, which was missing event flow information, protocol information and timing information, we will introduce timed Petri-nets, with two special places, which we describe below. By doing so we can express all the missing information.

- ⇝ *event flow information* will help in avoiding conflicts between blocking / non-blocking calls, signaling / polling / waiting architectures and syntactical problems.
- ⇝ *timing* information. By simulating the Petri-net in such a way that timers trigger certain places it might be possible to implement 'required' timing information'. By observing timers and detecting timeouts at 'offered' behavior, it is possible to detect errors.
- ⇝ *communication*: by introducing two places, a special receive-source place and a special send-sink place it might be possible to verify the correct working of the communication link. One place is a source-place to receive raw protocol data. The second is a sink-place that is used to send out raw data.

However, how we present these Petri-nets to the end-user is another problem. In this dissertation we have already created a Petri-net text format that is easy to use and can be expanded to more difficult Petri-nets offering different kinds of behavior. In this case we will need to add optional timing information on the arcs. Also, we cannot expect that the end-user exactly knows the underlying protocol (with the exact control data), therefore it might be a better choice to simply store the actual data in these places. Also because the use of buffers is an important issue in this case, it might be useful to introduce such an abstract data type into out Petri-nets. Also a necessity is the notion of a session, this will also be marked within the tokens. An example is given in figure 13.5.

13.7.6 Create the Adaptor

Once the missing information is present and the requirements are known, it is time to look for solutions. This involves finding out which requirements can result in a compilation of an adaptor, and which requirements require the insertion of a general algorithm. In both cases it is important to decide what kind of internal representation will be used. In this dissertation the internal representation of the liveness-module and the enforce-action module was similar to the Petri-net representation, however, the representation of the concurrency module was slightly different and more tuned toward its efficient working.

When using learning algorithms, make sure to use an automatic process to verify whether the representation is good. In the end, a computer needs to use the representation to (re)act correctly to certain situations. Therefore it is important to have a representation that quickly leads to a correct behavior. In other words: the bias of the representation should be measured and should be optimal. In this dissertation we have verified our Petri-net representation of the liveness-module by means of a genetic algorithm. E.g, the representation of the liveness module.

Once solutions have been identified to match the different requirements investigate how these solutions can be modularized. Afterward investigate what kind of topology is necessary to make the entire adaptor work. Important issues:

1. how can every requirement be satisfied ? compilation vs interpretation ?
2. when using learning algorithms: How to guarantee a correct bias ?
3. can certain requirements be modularized ?
4. what kind of topology will be used ?

Example: In our socket case, the implementation of the adaptor might be similar to the adaptor presented in this dissertation.

⇝ To guarantee a correct data flow, our adaptor can investigate at the application side which data it places in the output tokens. At the kernel/library side it can do a reachability analysis to deduce how to send data over the socket.

⇝ Solving timing information can be done by inserting a Petri-net evaluator that knows how to handle timing.

Probably the only two modules necessary to make this work is one module that communicates to the user level application and a second module that communicates to the actual library/kernel. The communication link between these two modules can be a common place, on one side understood as a receiving place, on the other side understood as a sending place. (figure 13.5 contains dotted transitions and lines which link them both together).
The evaluation of these socket nets should be able to handle the notion of threads: creating ones when necessary, managing incoming threads and destroying others. This can be done by using the 'session' knowledge present within the different tokens.

The above guidelines of course should be embedded in a typical prototype/experiment cycle. With these guidelines it should be possible to create adaptors for domains other than concurrency conflicts in open client server architectures. Essentially , any place where one interface is fitted with multiple implementations is a candidate for automatic adaptation.

13.8 Future Work

13.8.1 Meta-conflicts

An area that is not investigated in this dissertation are meta-conflicts. These are conflicts that occur between the different interface-descriptions. We have already touched this issue in section 12.7.1 on page 198. However, before the problem of meta-conflicts can be investigated, a large number of adaptors based on extra formal documentation needs to be available.

13.8.2 The problem of checkpoints

Test-scenarios

In our solution to the liveness problem we assumed that the programmer can easily specify checkpoints in the source code. However, if this is not possible, test-scenarios can be used. A test scenario specifies which traces are considered to be 'good'. Based on this information, rewards can be obtained. Illustration 13.6 shows two test-scenarios. Scenario 1 shows that the client will send out a `Lock` and expects a `LockTrue` in response. Afterward an `Act` will be sent out and an `ActDone` is expected to return. Finally an `Unlock` is sent out and an `UnlockDone` should return. The second scenario illustrates more or less the same, the only difference is that now two `Act`'s are sent out. The second scenario can be used to teach a learning algorithm that after the first `act` message multiple other `act` messages might be sent out. Given these two test-scenarios one can easily implement a tracker in the Petri-net that will assign rewards when appropriate.

If we go even a step further, it might be possible to convert a number of test-scenarios automatically to the Petri-net description of a component [EK98]. However such a process heavily relies on a correct generalization. Given the two scenarios in figure 13.6, one might assume that the component will also work for 3 or more acts at the same time. However, nothing ensures this is the case. Depending on the algorithm another generalization might occur.

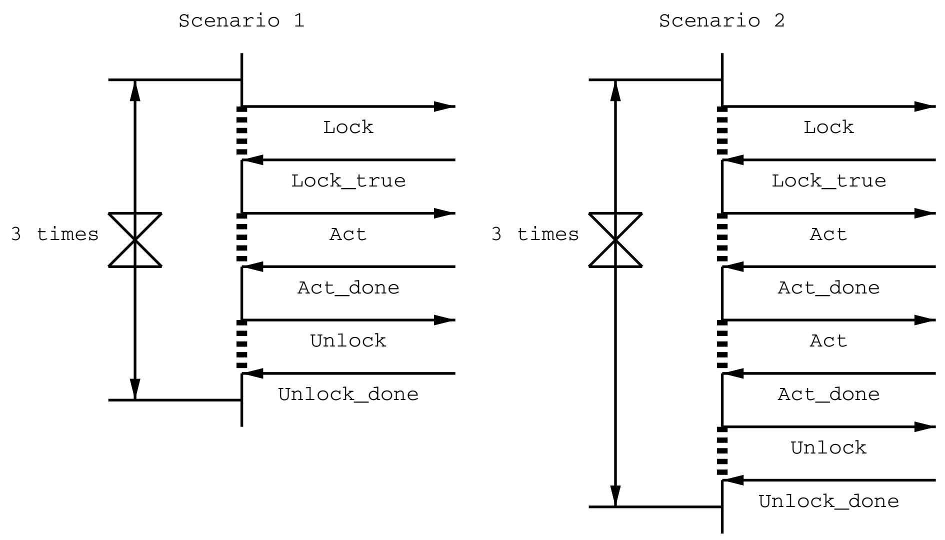


Figure 13.6: Two test-scenarios to verify the correct working of an adaptor.

Annotated Petri-nets

Instead of using Petri-nets and separate checkpoints, the developer of the Petri-nets could also tag arcs with a priority, which would resemble the rewards offered by the different components.

13.8.3 Other Learning Approaches ?

We have made the assumption that every future reward is statistically correlated to the current state of the Petri-net involved. This was necessary to demonstrate that the mapping of our liveness problem to a reinforcement learning problem was well defined. However, if we have made the assumption of a reward/marking correlation, we only did this from a practical point of view: for the programmer it is easy to specify checkpoints. In the conducted experiments we have observed that in general this did not form any problem, however a large scale investigation of this property would be very useful, especially to learn the boundaries of our approach.

We have already seen that if the reward is not entirely dependent on the locking strategy some oscillation in the Q values will occur, on the other hand if the reward is used to express a hidden property of the locking strategy it will not be able to learn it.

13.8.4 Peer to Peer Concurrency

In this dissertation we did not fully investigate how peer to peer concurrency could be managed. All the examples we have presented work with only one server. As explained in section 12.7.2 there are some major problems involved with peer to peer concurrency. The biggest problem of such an environment is that not all communication can be mediated and that different components might communicate indirectly with other components. To solve this problem adaptors themselves should be able to coordinate their behavior and learn how to behave to support an implicit present overall behavior. Theoretically, game theory might be applicable.

Game theory [Nas50b, Nas50a, Nas51, Nas53] requires agents that choose a discrete action. Depending on the action they choose they can either win or lose. The outcome of a game not only depends on the action chosen by one agent but on the interaction between all the chosen actions (this is typically the situation in peer to peer networks). A game is typically characterized by 5 elements:

1. the *players*, how many are there ? Is there a chance-player (or is there some random element) ? In our situation these could be the components and the delay times when sending messages.
2. a set of *possible actions* for every player. This largely depends on what kind of possibilities a component has. Probably this will be mapped onto the sending of a message or waiting with sending the message.
3. the *information* players have available when choosing their actions. It is clear that no adaptor in a peer to peer network will have all the information.
4. a *measurement* which describes the payoff for all combinations of actions. It should be possible to embody the overall required behavior into some kind of reward system.
5. a *description* of what every player tries to accomplish. These are the requirement of every agent. In this dissertation we have assumed that every agent tries to accomplish the same.

The 5 elements that characterize a game are present in typical peer to peer networks, so game theory might be a good start to investigate the problem of conflicting interfaces in peer to peer networks.

13.8.5 Learning an Interface Description

As explained in section 12.3, all investigated concurrency interfaces behave as simple finite state machines. Therefore it might be possible to deduce the behavior of an interface in an automatic way. For instance, a learning algorithm could, given an API of a component's interface, try out which actions are possible at which time and construct a Petri-net description of the interface automatically.

To do so a program ρ, that is supposed to learn a Petri-net description of component α, which offers only a syntactical interface description δ, could follow different strategies:

1. *learn hidden variables* of α by only accessing δ. The problem of modeling the behavior of α by only looking at δ is essentially finding out which variables are useful to describe the behavior of α over δ. The problem is that all too often these variables are hidden within the semantics and not actually present at δ. Therefore two ideas might help
   (a) *observe the behavior* of a standard communication and find out hidden variables by statistical analysis of the message flow.
   (b) *generation of test*-messages by ρ to validate and/or test certain variables:
      i. useful to verify whether a hidden variable is truly a distinct state and not merely a coincidence.
      ii. useful to check the boundaries of variables.
2. directly *observe the component* α. Instead of looking for hidden variables it might also be possible to directly investigate the behavior of the component.
   (a) *observing its binary state* before and after handling a message. This avoids the use of looking for hidden variables but can easily create too large models and describe states that are entirely redundant to model the behavior of δ.
   (b) *observe the control flow* within the program when handling a message. This should make it possible to detect whether all branches of the component have been investigated and possible how other branches can be investigated. This might help in creating the expressions on the arcs of the Petri-net.

3. *model generation & verification*. Generate a random model γ. Afterward ρ can verify, by checking consistency and completeness whether γ is a correct model of α. To do so, genetic algorithms/programming might be useful (given the correct bias of course). After verification of a model γ, crossover and mutation could help in refining and creation of a better model.

4. *Petri-net representation* in such a way that

   (a) the Petri-net it is not too small. For instance, one place that captured the entire state of the component, with 5000+ transitions describing every possible message is clearly not a good Petri-net

   (b) the Petri-net is not too large. It is also useless to create a place for every memory-cell available to the component.

   (c) the representation should only contain places to model a certain requirement such as i) when a certain message can be send by only looking at the client, or ii) when a certain message can be send, looking at the server, or iii) to capture the state of a component such that it can be rolled back at a later time, or others...

   (d) automatic reduction of the Petri-net might help in removing obsolete states, not necessary to model δ.

The above are some ideas about how one might try to learn the behavior of an interface automatically. Whether it is truly possible to learn the behavior of an interface automatically remains an open question. If we would be able to do this, then the impact of our work would be substantially greater, because then we would be able to mediate differences between components in an even more automatic way. As a trade-off, the advantages of using a formal interface specification would be lost. We would no longer be able to verify the completeness and consistency of an interface description.

13.8.6 Determinism versus Non-Determinism

This dissertation focuses on the generation of intelligent adaptors. We made the assumption that we have the concurrency strategies specified as a Petri-net. From a pragmatic point of view, this is very useful because it offers the developer all kinds of interesting information. However, the *essential* reason why we need Petri-nets is to introduce determinism and avoid that the liveness module could bring the client components in an invalid state by sending out a wrong message. E.g., if the client component does not know that before the first `setPosition`, a `joinActor` needs to be sent, the client component might fail to work properly.

In the enforce-action module we also need a similar deterministic property. This module needs to *know* how it can bring the server component in a certain state. Here we assume that the server is well written and that every possible synchronization action that can be invoked upon the server can easily be undone. Of course, to avoid denial of service attacks and to increase its robustness, every good server offers this kind of functionality. However, in general, the concurrency module needs be sure that it can *deduce deterministically* what to do and that it will *always* be able to reach a certain state.

Within the concurrency module a similar *deterministic* problem occurs. However this problem cannot be solved easily. The concurrency module resolves race conditions. However, it is unable to resolve deadlocks when it detects them. The reason is that it is unable to *anticipate* a deadlock, because it has no knowledge of it, and that, once it has detected a deadlock it is unable to undo previous actions on the involved components.

From these three modules we see how *required determinism* poses problems. During this dissertation we have tried to generate adaptors that work without making mistakes that cannot be revised afterward. The reason why we imposed such requirement is that all real-world computation happens in a *controlled non-deterministic* (hence deterministic) way. If two components communicate, there are *exactly* two components and the functionality offered by those two components is *exactly* understood. If an adaptor wants to mediate the differences between those components it should be *deterministically* correct, otherwise it would *certainly* bring disaster upon one of both (if not both) components. This deterministic view on computation is an illusion, for in open distributed there are many opportunities when a component might fail; all too often the reasons lie within uncontrolled upgrades or unreliable infrastructure. Nevertheless developers still hold on to an illusion of determinism and construct programs that behave completely deterministically.

In this dissertation we made exactly the same assumption. However, one can also approach the problem from the opposite side. One might be tempted to embrace the *non-determinism* found in open distributed systems and investigate how different but similar working components might *emerge* behavior that satisfies certain requirements.

By exploiting non-determinism one might create programs, which act creatively and intelligently, because they would be truly able to find and apply clues in the environment which a deterministic process cannot see. This kind of research however is still barely started and certainly not available in the field of 'services' offered on the Internet. Some pointers might give a lead such as swarm intelligence [KRCE01], amorphous computing [HAAC$^+$00] and cellular automata [TM87].

Appendix A

Thread Based Models for Distributed Systems

JAVA WAS ORIGINALLY CONCEIVED BY SUN MIRCOSYSTEMS to be used on embedded systems. However, it turned out to be a better language for the Internet because it offered a virtual machine that could run Java applications on all kinds of different architectures. Java is an object oriented programming language that supports automated memory management (= garbage collection), threads, and interfaces. Libraries like Java RMI (remote method invocation) are available to support semi transparent remote accessibility. This model, a typical thread based, transparent distribution model, is the focus of this appendix. We will investigate the suitability of Java and its thread based model for writing adaptors. This chapter explains why we opted for an event based model. It gives also isnight into the problems of creating adaptors for such models.

A.1 Naming/Finding services

THE FIRST THING EVERY distributed application has to do before connecting a remote object, is looking up what object it should connect to. For this purpose Java RMI uses a special server which should be started on the machine where remote objects will be exported. This server is called the *RMI registry*.

If an object wants to export itself it can use the `Naming` class to export its own name to the registry. From then on all applications can ask the registry a reference for the object with that name.

It is clear that such a mechanism is very rudimentary, because we still need to know the name of the machine where the registry (and the objects as such) are running. To support new technologies, such as wireless embedded devices, Sun developed JINI, which allows peers (clients) to look up other peers (servers) using a description of the required capabilities, instead of a simple name. The initial lookup to find a JINI directory server is done with a broadcast.

A.2 Communication

JAVA PROCESSES can communicate by using Java RMI. Note that Java RMI can only communicate between Java Processes, which is a major drawback of RMI.

Sun's serialization and deserialization interface to Java [CFKL91] helps with exporting an object graph to a byte stream. The standard behavior for serialization is to serialize the object and all the objects it contains. If we want to modify this behavior we have to implement an *externalizable* interface which describes how the object is to be exported.

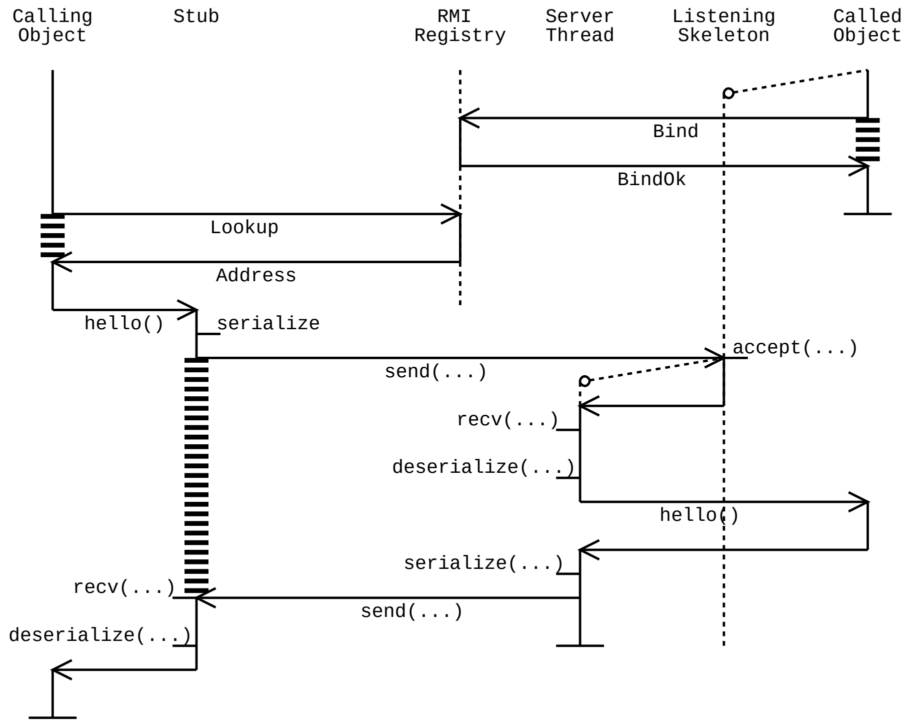


Figure A.1: Remote object calling in Java RMI.

To be able to contact a remote object, as if it were a local object, one should create stubs for all the remote objects which will be contacted.[1] Such a stub will implement all the methods the remote object supports and filling in the bodies with code which contacts the remote object. The logic of such a stub body is quite simple:

1. *Contact* the remote object using sockets
2. *Serialize* all the arguments passed to the stub by the caller.
3. *Send* out all arguments
4. *Wait* for an answer, which will be the result of the remote method call.
5. *Receive/read* the result
6. *Deserialize* the result
7. *Return* the result to the original caller

The object that is being exported at the server side, offers a way to be contacted by remote clients by means of a skeleton. This skeleton contains some listening code and logic to contact the actual exported object. The logic in a skeleton is as follows:

1. *Listen* for connections
2. *Accept* an incoming connection
3. *Receive* the serialized arguments

[1]This can be automatically done using *rmic*.

4. *Deserialize* the arguments
5. *Invoke* the method call upon the correct object
6. *Serialize* the result of the invocation
7. *Send* back the result over the socket

Figure A.1 illustrates how we can transparently contact a remote object. We see how the setup is inherently client-server. When we want to contact a client from within the server the client needs to export an interface and become a listening server itself.

A.3 Openness & Remote Objects

ANOTHER POINT TO MAKE with respect to stubs and skeletons is that they are generated by a compiler, called *rmic*. This means that stubs and skeletons are created at compile time. As a direct consequence we cannot easily communicate with an *unknown* process at runtime. In the setting of this dissertation, this is not acceptable because we don't know the interface we will link to.

One could think of a solution by creating stubs and skeletons as needed: any time a connection to another machine is needed, the interface description could be downloaded from the server and compiled into a stub class that would connect to the appropriate skeleton at the server. This compilation phase would require a compiler or Java byte code assembler, and would consume a lot of time for simply setting up a link to a remote object. It is clear that this is not practical and not a good approach at all.

Another approach, as used by Smalltalk [AGR83] users, is using the meta-level interface and overriding a method such as `doesNotUnderstand`. With this a simple and general stub could be created. The only method of the stub would be the `doesNotUnderstand`. This method would be called every time an undeclared method is invoked upon the object as can be seen in figure A.2. The `doesNotUnderstand` in turn would look at the actual method invocation and pass it along to the skeleton. However, as it turns out, this is impossible with Java because the meta-level interface of Java is not strong enough.

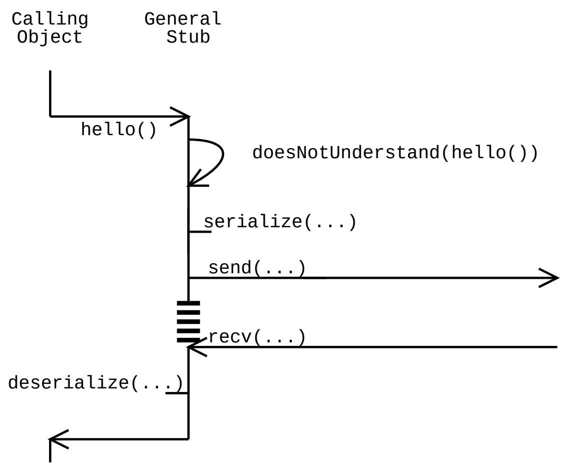


Figure A.2: The behavior of a general stub, using `doesNotUnderstand`.

A.4 Java Threads

IN MANY ASPECTS JAVA is an innovating language. One of these innovations is the introduction of a standard threading library.[2] A thread is an execution context which can run together with other threads in the same environment. The difference with processes is that threads do share memory, while processes don't share memory.

Algorithm 27 A distributed recursive factorial.

```
public class FacApp extends UnicastRemoteObject
  implements FacCalcer
{ ...
  public int fac(int n, FacCalcer fc) throws RemoteException
    {
    if (n==0) return 1;
    return n*fc.fac(n-1,this);
    }
  public static void main(String args[])
    {
    FacCalcer fc;
    FacApp self=null;
    try {
      int answer;
      self=new FacApp();
      try {
        fc=(FacCalcer)Naming.lookup("faccalcer");
        answer=fc.fac(3,self);
        System.out.println("The answer my friend ... "+answer);
        }
      catch (NotBoundException e) {
        System.out.println("faccalcer bound");
        Naming.bind("faccalcer",self);
        }
      }
    catch (Exception e){...}
    }
}
```

The availability of threads in Java is of crucial importance for the internal workings of Java RMI as we will explain now. Java RMI, as already seen, waits before returning an answer: a client can ask the server to execute a method and return the answer. In the meantime the client simply waits. Now, let's have a look at the Java program in algorithm 27 (page 230).

The `FacApp` class exports a `FacCalcer` interface. A `FacCalcer` is an object that calculates the factorial of a certain number by performing a recursive call. To do the recursive call the `FacCalcer` needs to receive another `FacCalcer` as can be seen in the definition of the `fac` method. When such a `FacCalcer` starts, it either becomes the master `FacCalcer` (by binding itself in the registry) or a slave `FacCalcer`, which will initiate the calculation of a factorial.

To start this program, first an *rmiregistry* should be running and afterwards two instances of the `FacApp` should be started. The last `facapp` (`faccalcer2` from now on) requests the first `faccalcer1` to calculate the factorial of 3. In return `faccalcer1` will request `faccalcer2` to calculate the factorial of 2... But how is this possible ? How can the original requester be interrupted while he is still waiting ? The answer lies in the Java threading mechanism. Every time an RMI call comes in, the server-thread will spawn a new thread which handles the request. This can be seen in figure A.3.

[2]compare this with C which has all kinds of dirty libraries and tricks like `setjmp` and `longjmp` to handle multiple sessions.

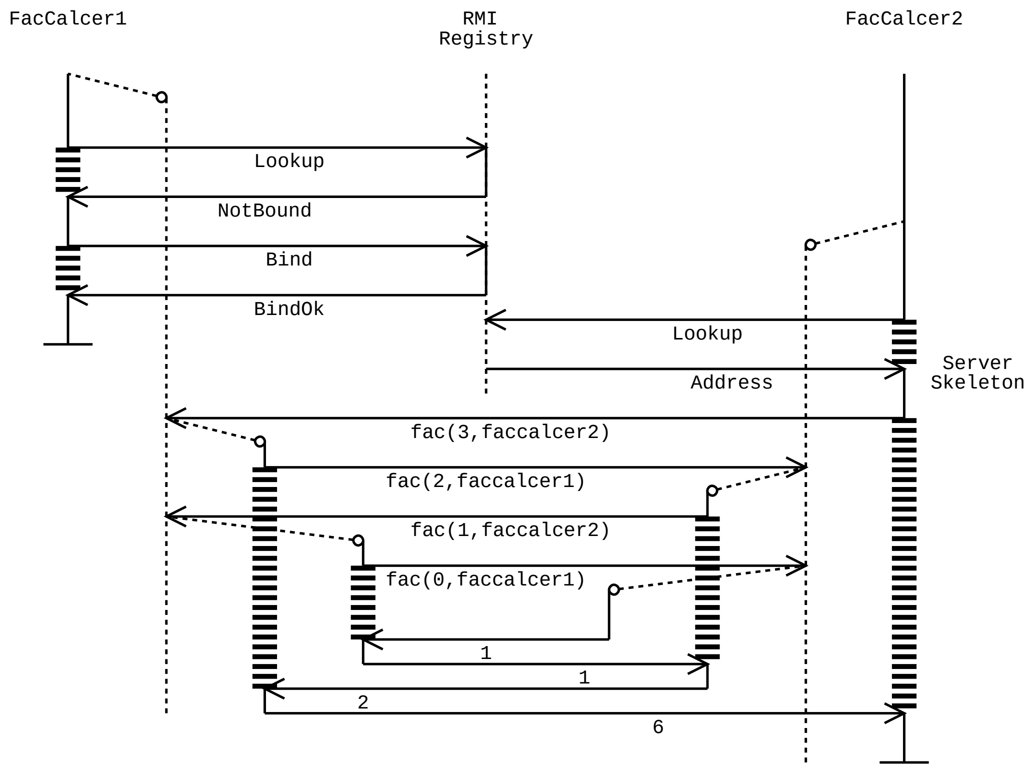


Figure A.3: In the above figure, full lines indicate a running process, dotted lines indicate a waiting (listening) process and horizontal dashed lines indicate a 'wait for return'. In this figure we see how a new thread is spawned every time a call comes in.

The result of this behavior is nicely what a programmer would expect. The problem is that this application suffers from a number of conceptual problems, which are essentially grounded in the inherent problems of distributed systems: concurrency and partial failure. We will explain this below.

A.5 Error handling: Exceptions

LET'S HAVE A LOOK at how failures of the underlying network and failures of processes are caught. Java has a well-known language construct of *exceptions* and uses this to report errors that occur when contacting a remote object. Technically, RMI achieves this by letting the stub throw an appropriate exception. When, on the other hand, the skeleton fails while executing the incoming message (because the program throws some kind of exception) it will simply serialize the exception and send it back to the client.

Although, this solution looks nice, there is not much that can be done when such an exception is caught. Do we reconnect with the server, do we inform the user, or what should we do ? The main problem programmers encounter here is how to handle those exceptions in a structured way.

To illustrate these problems, suppose we have a process A that calls another process B, which in its turn will contact process A again. What will happen when process B dies at the moment A is sending back its result to B ? Will process A know in what state T_0 is ? How will the exceptions cascade ? Figure A.4 illustrates this. At the moment T_1, i.e. at the moment process A received a broken pipe exception from the underlying socket layers, it does not know anymore that it originally received a call from process B. Process A cannot easily know that an internal thread T_0 is still executing or not. The net result is that process A ends up to be in an unknown, probably invalid, state.

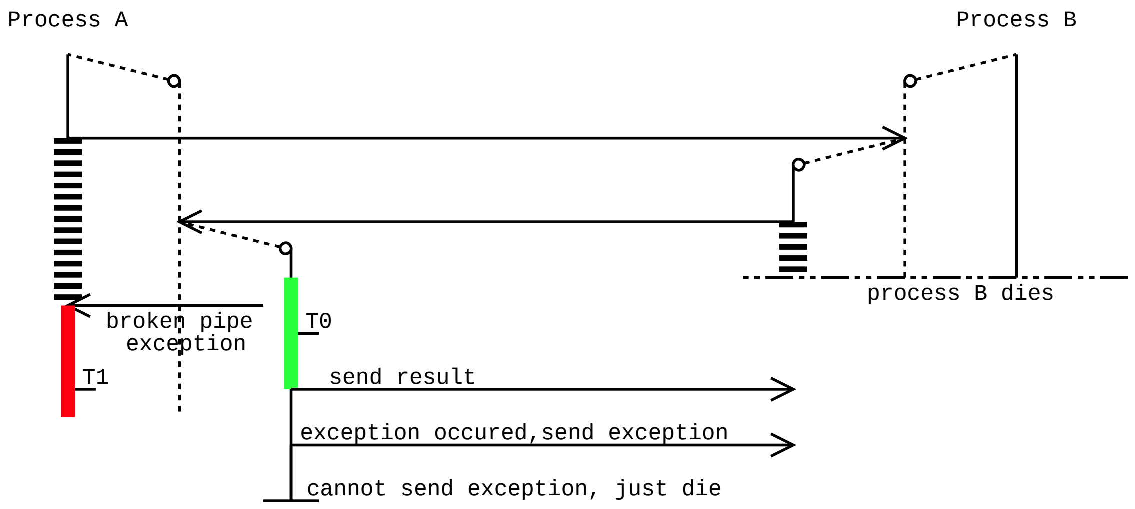


Figure A.4: An illustration what can go wrong with the nested calling conventions of Java RMI. At the moment process B dies, process T_1 does not know whether T_0 has already been executed or not. This leaves A in an unknown state.

A.6 Java RMI and Concurrency

A.6.1 Concurrency Primitives

SINCE JAVA IS A LANGUAGE with native support for threads, we need to investigate how concurrency can be managed and what kind of language constructs are available. The first and most important language construction is *synchronized*.

```
synchronized(foo)
  {
  int val = foo.read();
  foo.write(val+1);
  }
```

In Java all objects can have a lock, when an object is synchronized the current thread will try to obtain a lock on that object, and if the lock is obtained, the statement block will be executed. When leaving the block statement the lock is released again. The object locks are reentrant, so the same thread can lock the same object multiple times. With this construct one can easily implement a critical section. However, it is still allowed that other accesses to the `foo` object are not synchronized, thereby ignoring concurrency behavior.

A second construct is the possibility to synchronize methods in an object. For example:

```
synchronized public void increase()
  {
  ...
  }
```

Which means that the increase method will only execute when the *this* object is locked. In fact we can write exactly the same as follows:

```
public void increase()
  {
  synchronized(this)
    {
    ...
    }
  }
```

Now, although a nice construct it suffers the same problems as all concurrency primitives in object oriented languages: *the inheritance anomaly*. Suppose, we specify a method in a class to be synchronized, this means that all overriding methods must be synchronized too. This effectively means that a subclass *cannot* choose to be not synchronized for its own actions, and be synchronized for the super calls. Aside from this annoying problem, there are a lot of other problems with respect to synchronization and concurrent object oriented languages.

Note that, aside from these (relatively low level) synchronization mechanisms, there are other solutions like wait-notify mechanisms, the Java Transaction interface which offers a much more high level approach to concurrency strategies and others. For a more detailed discussion about concurrency management and Java see [Lea00].

A.6.2 Java RMI

THE JAVA THREADING MECHANISM and the way Java RMI uses it, makes it possible for one remote object to be invoked multiple times by different threads on a concurrent basis. This means that the object's state will be soon invalid if we don't guard access to the remote method.

If we now use the standard Java keyword `synchronized` to guard access to the remote object, we see that only one thread can enter the remote object at a time, thereby placing all other threads in wait until the object becomes available again. However, this still raises some problems.

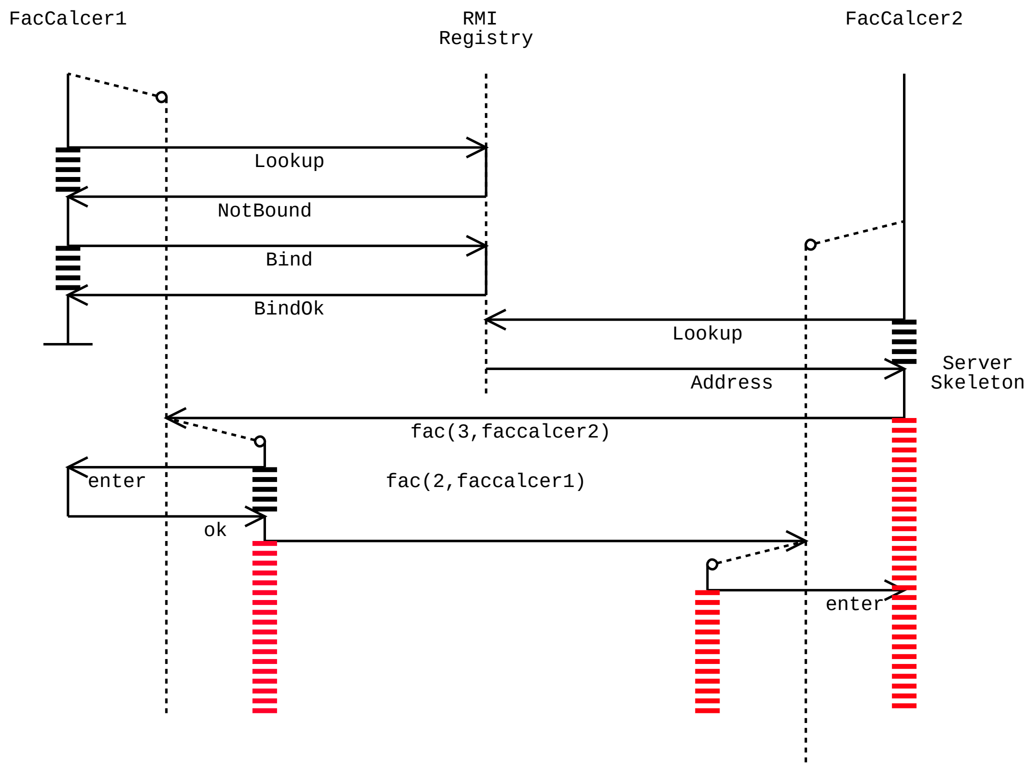


Figure A.5: In the above figure we see how a deadlock occurs between two synchronized RMI calls, which normally would work if not distributed.

For example, let us take the previous factorial example and assume that the `fac` method is synchronized. What will happen ? One would expect the program to produce its standard result 3,2,1,... and so on. In practice this will not happen because the thread that comes back from the first `faccalcer` is different from the thread that is waiting inside a synchronized block. So this new thread will have to wait for the lock to be released and finally deadlock because this new thread is supposed to offer an answer before the calling thread will release the lock. In figure A.5 we see the control flow of this program.

This example illustrates that we still need some active form of session management in distributed systems. It also illustrates how concurrency problems cannot simply be solved by synchronizing methods. In distributed systems every remote interface will need to export some kind of concurrency interface, and the implementation will need to have a well thought off concurrency management strategy. This concurrency management is typically larger than the actual actions to perform. We will come back on these issues in detail in chapter 5.

A.7 Writing Adaptors

WE HAVE NOW SEEN how Java RMI works. We have seen how concurrency is managed and how threads are used. Writing adaptors with Java RMI is clearly not as easy as with the component system we have been using.

1. The Java RMI call-wait-return calling conventions makes implementing an adaptor difficult. An adaptor can receive multiple incoming calls and will spawn multiple internal threads in response. These threads need to be guarded somehow. Writing this guarding mechanism is not trivial since the threads themselves are implicitly started.

2. The Java RMI registry makes it difficult to plug in an adaptor between two communicating processes. When one process contacts another it lookup the name of the remote object. If we could rebind this name to point to the address of an adaptor this could be possible. However it is impossible to rebind a name (let us say `Server`) to a new location (`Adaptor`). This is necessary to make sure that all processes wanting to contact the original `Server` will contact our `Adaptor`.

3. A typical Java RMI connection only sends data and receives one answer. If we want to adapt data sent over connections between different processes, our adaptor will need to implement both directions. It will need to listen to the first process, as well as to the second process. All incoming connections must be adapted and identification of the correct remote object should be present. An important issue here is that we don't know beforehand which objects are being exported by a Java Process and whether they should be adapted or not.

4. At runtime we cannot contact a process unknown at compile time because the Java RMI reflection interface is not good enough to be able to make generic stubs.

5. The concurrency guarding implemented in a remote object is not visible to the outside world, nevertheless it has a strong impact in how the application works and will respond to incoming calls. The adaptor should work together with the synchronization behavior of client and server. But since the concurrency interface is not exported this cannot be achieved.

Bibliography

[AdFDJ03] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan, *An Introduction to MCMC for Machine Learning*, Machine Learning **50** (2003), 5–43.

[AGR83] Adele Goldberg and Dave Robson, *Smalltalk-80: the language*, Addison Wesley, 1983.

[ALSN01] Franz Achermann, Markus Limpe, Jean-Guy Schneider, and Oscar Nierstrasz, *Piccola – a Small Composition Language*, Formal Methods for Distributed Processing – a Survey of Object-Oriented Approaches (Howard Bowman and John Derrick, eds.), Cambridge University Press, 2001, pp. 403–426.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott, *A Foundation for Actor Computation*, Journal of Functional Programming **7** (1997), no. 1, 1–72, **Extensive formal explanation (72 pages) of Actor Systems. The 'become' operator is one of the most difficult to understand.**

[BBC] Andrea Bracciali, Antonio Brogi, and Carlos Canal, *Systematic component adaptation*, **This paper introduces a CSP-like formalis to describe the behavior of interface. The same notation is used to express adaptors and once this is done it is possible to verify twhether the requirements are matched within the adaptor.**

[BBT01] Andrea Bracciali, Antonio Brogi, and Franco Turi, *Coordinating interaction patterns*, Selected Areas in Cryptography, 2001, pp. 159–165.

[BFDV01] Werner Van Belle, Johan Fabry, Theo D'Hondt, and Karsten Verelst, *Experiences in Mobile Computing: The CBorg Mobile Multi-Agent System*, Proceedings TOOLSEE 2001, Zurich (http://borg.rave.org/) (Wolfgang Pree, ed.), vol. 38, IEEE Computer Society Press, March 2001, **Summary of our experience with the self made Borg Mobile Multi Agent System.**, pp. 1–9.

[BFF$^+$95a] Eike Best, Hans Fleischack, Wojciech Fraczak, Richard P. Hopkins, Hanna Klaudel, and Elisabeth Pelz, *A Class of Composable High Level Petri Nets with an Application to the Semantics of $B(PN)^2$*, Application and Theory of Petri Nets (Giorgia De Michelis and Michel Diaz, eds.), Lecture Notes in Computer Science, vol. 935, Springer Verlag, 1995, **Describes a set of high level operations to compose petri-nets.**, pp. 103–120.

[BFF$^+$95b] Eike Best, Hans Fleischhack, Wojciech Frączak, Richard P. Hopkins, Hanna Klaudel, and Elisabeth Pelz, *An M-Net semantics of $B(PN)^2$*, STRICT (Jörg Dessel, ed.), Lecture Notes in Computer Science, Springer Verlag, 1995, **Describes the transformations necessary to compile $B(PN)^2$ programs to High Level petri-nets, called M-Nets.**, pp. 85–100.

[BG98] Eike Best and Bernd Grahlmann, *Format Descriptions*, Tech. report, Institut fur Informatik Marienburger Plaz 22 D-31141 Hildesheim, 1998, **Internal formats used by the PEP Tools**.

[BH93] Eike Best and Richard P. Hopkins, *B(PN) – a Basic Petri Net Programming Notation*, Proceedings PARLE'93 (A.Bode, M. Reeve, and G. Wolf, eds.), Lecture Notes in Computer Science, vol. 694, Springer Verlag, 1993, **A process algebra designed to map easily on petri nets.**, pp. 379–390.

[BK01] Meredith Beveridge and Philip Koopman, *Jini meets embedded control networking: a case study in portability failure*.

[BL01] Joel Brinkley and Steve Lohr, *U.S. v. Microsoft*, McGraw-Hill, ISBN: 0 07 135588-X, 2001, **Stunning facts how Microsoft crushes opponents and how nobody seems to care.**

[BMAPY97] Marius Bozga, Oded Maler, Amir Pnueli, and Sergio Yovine, *Some Progress in the Symbolic Verification of Timed Automata*, Proc. 9th International Conference on Computer Aided Verification (CAV'97) (O. Grumberg, ed.), vol. 1254, Springer Verlag, 1997, **Describes how BDD (Binary Decission Diagrams) can be used in certain domains for a faster timing verification of large Petri Nets.**, pp. 179–190.

[BOP] R. Bastide, O.Sy., and P. Palanque, *Formal specification and prototyping of corba systems*, Proceedings of ECOOP'99, Spring Verlag LNCS 1628, **Uses Petri-nets to describe the behavior of CORBA objects.**, pp. 474–494.

[BP96] Eike Best and Ctuscia Palamidessi, *Linear Constraint Systems as High-Level Nets*, CONCUR (Ugo Montanari and Vladimiro Sassone, eds.), vol. 1119, Springer Verlag, 1996, **Makes the mapping from linear programming to timed Petri-nets. Also proves that constructing a reachability graph of timed Petri-Nets is NP-complete.**, pp. 498–513.

[Bra01] P. Brada, *Towards Automated Component Compatibility Assessment*, Presented at Component Workshop ECOOP 2001, Budapest, Hungary (2001), **Stresses the importance of component compatibility guarantees. However, implicit requirements are not addressed.**

[Bro97] J. G. Brookshear, *Computer science: An overview*.

[CD99] Juan Carlos Cruz and Stéphane Ducasse, *Coordinating Open Distributed Systems*, Proceedings Workshop on Future Trends of Distributed Computing Systems, vol. 7, IEEE Computer Society Press, December 1999.

[CDGM97] A. Chavez, D. Dreilinger, R. Guttman, and P. Maes, *A Real-Life Experiment in Creating an Agent Marketplace*, Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, London, UK, April 1997.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg, *Distributed Systems, Concepts and Design*, Addison Wesley, 1994, **Chapter 12 covers transactions, chapter 13 covers concurrency control and chapter 14 covers distributed transactions**.

[CFKL91] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim, *Object Serialisation for Marshalling Data in a Java Interface to MPI*, Java Grande, 1991, **Among other things shows benchmarking results of Java Serialisation**, pp. 66–71.

[CFL$^+$99] R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Sobroff, James Mayfield, and Akram Boughannam, *An agent-based infrastructure for enterprise integration*, First International Symposium on Agent Systems and Applications (ASA'99/Third International Symposium on Mobile Agents (MA'99), 1999.

[CFP$^+$01] C. Canal, L. Fuentes, E. Pimentel, J.M.Troya, and A. Vallecillo, *Extending CORBA Interfaces With Protocols*, The Computer Journal **44** (2001), no. 5, 448–462, **Argues that an extension to IDL's might be appropriate.**

[CG] Craig. M. Chase and Vijay K. Garg, *Detection of Global Predicates: Techniques and their Limitations*, Tech. report, Parallel and Distributed Systems Laboratory, Electrical and Computer Engineering Department, The University of Texas.

[CL94] Robert Cyphner and Eric Leu, *Repeatable and portable message-passing programs*, 22–31.

[CP95] Stephen Cranefield and Martin Purvis, *Agent-based integration of general-purpose tools*, Proceedings of the CIKM'95 Workshop on Intelligent Information Agents (Baltimore, Maryland) (Tim Finin and James Mayfield, eds.), 1995.

[Cro96] J. Crowcroft, *Open distributed systems*, UCL Press, 1996.

[Dar59] Charles Darwin, *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*, 24 November 1859, **The entire text of this historical work can be found on http://www.literature.org/authors/darwin-charles/the-origin-of-spieces/**.

[DBS$^+$01] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.C. Burgelman, *Scenarios for Ambient Intelligence in 2010*, IPTS-Sevile, see http://www.cordis.lu/ist/istag.htm (2001), **A set of nice written scenarios wrt ambient intelligence.**

[D'H95] Theo D'Hondt, *Pico*, http://pico.vub.ac.be/ (1995), **Pico is an educational language used at the VUB. This is the website.**

[Dic00] C. Dichev, *Advanced operating systems lecture notes*.

[DK96] Dimitirs C. Dracopoulos and Simon Kent, *Bulk Synchronous Parallelisation of Genetic Programming*, Applied Parallel Computing: Industrial Strength Computation and Optimisation, Proceedings of the third International Workshop PARA'96 (Berlin, Germany) (Jerzy Waśniewski, ed.), Springer-Verlag, 1996, pp. 216–226.

[Edw99] W. Keith Edwards, *Core JINI*, ISBN: 0-13-014469-X, The Sun Microsystems Press Java Series, Prentice Hall PTR, Upper Saddle Rive, NJ 07458, 1999, **Explains the reasons behind and the implementation of Jini.**

[EK98] Mohammed Elkoutbi and Rudolf K. Keller, *Modelling Interactive Systems with Hierarchical Coloured Petri Nets*, Proceedings of the Conference on High Performance Computing (1998), **Given a set of UML usage scenarios, a Petri-net is generated.**

[EN94] Javier Esparza and M. Nielsen, *Decidability Issues for Petri Nets*, Bulletin of the EATCS, vol. 52, February 1994, **A good introduction to the decidability issues of Petri-nets**, pp. 115–129.

[ERV96] Javier Esparza, Stefan Romer, and Walter Vogler, *An Improvement of McMillan's Unfolding Algorithm*, Int'l Proceedings of Tools and Algorithms for Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 1055, Springer Verlag, 1996, **Explains the reason behind Petri-net unfolding, their limitations and improves McMillans original algorithm.**, pp. 87–106.

[ES] Michael Erdmann and Rudi Studer, *Ontologies as Conceptual Models for XML Documents*, Tech. report, Institut fur Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) University of Karlsruhe (TH) D-76128 Karlsruhe (Germany) {erdmann, studer}@aifb.uni-karlsruhe.de.

[FDF03] R. Figueiredo, P. Dinda, and J. Fortes, *A case for grid computing on virtual machines*, Proceedings International Conference on Distributed Computing Systems (ICDCS) (2003).

[FFMM94] T. Finin, R. Fritzson, D. McKay, and R. McEntire, *KQML as an agent communication language*, Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94) (Gaithersburg, MD, USA) (N. Adam, B. Bhargava, and Y. Yesha, eds.), ACM Press, 1994, pp. 456–463.

[FG98] Hans Fleischhack and Bernd Grahlmann, *A Compositional Petri Net Semantics for SDL*, ICATPN (J.Desel and M. Silva, eds.), Lecture Notes in Computer Science, vol. 1420, 1998, **Describes how SDL can be converted to M-Nets by means of composition of a number of basic building blocks.**, pp. 144–164.

[FKK95] Bause Falko, Peter Kemper, and Pieter Kritzinger, *Abstract Petri-Net Notation*, Petri Net Newsletter **49** (1995), 9–27, **Describes an abstract Petri-net notation which is suitable for TEX. Otherwise quite unreadable and probably not suitable as a Petri-net syntax.**

[Fla94] Peter Flach, *Simply Logical, Intelligent Reasoning by Example*, ISBN: 0-471-94152-2, Wiley Professional Computing, 1994, **Interesting book that explains how 'intelligent' programs can be written in prolog. Covers a lot of examples and a lot of different approaches/techniques.**

[G.01] Booch G., *Object Oriented Analysis and Design with Applications*, Santa Clara, California: The Benjamin/Cummings Publishing Company, Inc., 2001.

[Gar97] Vijay K. Garg, *Observation and Control for Debugging Distributed Computations*, 3rd international workshop on Automated Debugging (AADEBUG97) (Linköping, Sweden), 1997, pp. 1–12.

[GF93] Michael R. Genesereth and Richard E. Fikes, *Knowledge interchange format version 3.0 reference manual*, Tech. report, Computer Science Department Stanford University Stanford, California 94305, 1993.

[GK99] Günther Kniesel, *Type-Safe Delegation for Run-Time Component Adaption*, Lecture Notes in Computer Science **1628** (1999), 351.

[Gol89] David E. Goldberg, *Genetic Algorithms in Search, Optimisation and Machine Learning*, ISBN: 0-201-15767-5, 1989.

[Gra] Bernd Grahlmann, *PEP: A Programming Environment based on Petri nets*, Institut fur Informatik Marienburger Plaz 22 D - 31141 Hildesheim.

[Gre65] Mendel Gregor, *Versuche über Pflanzen-Hybriden*, March 1865, **The entire text (also translated to English) of this historical work can be found at http//www.mendelweb.org/**.

[Gru93] Thomas R. Gruber, *Towards Principles for the Design of Ontologies Used for Knowledge Sharing*, Formal Ontology in Conceptual Analysis and Knowledge Representation (1993).

[GW93] V. Scott Gordon and Darrell Whitley, *Serial and Parallel Genetic Algorithms as Function Optimisers*, Proceedings of the Fifth International Conference on Genetic Algorithms (San Mateo, CA) (Stephanie Forrest, ed.), Morgan Kaufman, 1993, pp. 177–183.

[HAAC$^+$00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Redhika Nagpal, Erik Rauch, Gerald J. Sussman, and Ron Weiss, *Amorphous Computing*, Communications of the ACM **43** (2000), no. 5, 74–82.

[Har87] D. Harel, *A Visual Formalism for Complex Systems*, Science of Computer Programming 8 **3** (1987), 231–274.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, International Series in Computer Science, Prentice Hall, ISBN: 0 13 153271, 1985, **The reference on CSP.**

[Hol] J.H Holland, *Escaping Brittleness: The Possibilities of General Purpose Learning Algorithms applied to Parallel Rule-Based Systems*, Machine Learning **2**, 593–623.

[HWSS95] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld, *Strongly Typed Genetic Programming in Evolving Cooperation Strategies*, Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95) (Pittsburgh, PA, USA) (L. Eshelman, ed.), Morgan Kaufman, 1995, pp. 271–278.

[JB99] Jan Bosch, *Superimposition: A component Adaptation Technique*, Information and Software Technology **41** (1999), no. 5, 257–273.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, *Object Oriented Software Engineering. A Use Case Driven Approach*, 1992.

[JDA97] Ellsberger J., Hogrefe D., and Saram A., *SDL, Formal Object-Oriented Language for Communicating Systems*, London: Prentice Hall, 1997.

[Jen94] K. Jensen, *An Introduction to the Theoretical Aspects of Coloured Petri Nets*, A Decade of Concurrency (J.W. de Bakker, W.P. de Roever, and G. Rozenberg, eds.), Lecture Notes In Computer Science, vol. 803, Springer Verlag, 1994, **Very good introduction to coloured petri nets. A must read for everyone involved with petri-nets. Also covers formal analysis of Petri Nets.**, pp. 230–272.

[JMW$^+$91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., *Object–Oriented Modelling and Design*, Prentice–Hall, 1991.

[Jør] Jens Bæk Jørgensen, *Construction of Occurrence Graphs with Permutation Symmetries Aided by the Backtrack Method*, Tech. report, Computer Science Department, University of Aarhus, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark, **Explains how to detect self-symmetries during reachability analysis by means of backtracking. However, not a published paper, it is a very interesting read.**

[KCJ98] Lars M. Kristensen, Soren Christensen, and Kurt Jensen, *The Practitioner's Guide to Coloured Petri-nets*, International Journal on Software Tools for Technology Transfer (1998), 98–132, **This paper provides a comprehensive road map to the practical use of CPNets and the Design/CPN tool. The paper is self-contained and does not assume any prior knowledge of Petri nets.**

[Kie96] T. Kielmann, *Designing a Coordination Model for Open Systems*, Proc. 1st International Conference on Coordination Models and Languages (Cesena, Italy) (P. Ciancarini and C. Hankin, eds.), vol. 1061, Springer-Verlag, Berlin, 1996, pp. 267–284.

[KLAPM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore, *Reinforcement Learning: A Survey*, Journal of Artificial Intelligence Research **4** (1996), 237–285.

[Koz92] John R. Koza, *Genetic programming; on the programming of computers by means of natural selection*, ISBN: 0-262-11170-5, The MIT Press, 1992.

[KRCE01] James Kennedy and with Yuhui Shi Russel C. Eberhart, *Swarm Intelligence*, ISBN: 1-55860-595-9, Morgan Kaufman Publishers, 2001, **Covers a) philosophical aspects of evolutionary computing and b) the algorithmic of co-evolving individuals.**

[KV97] Ekkart Kindler and Tobias Vesper, *A Temporal Logic for Events and States in Petri Nets*, Proceedings of the Workshop "Petri Nets in System Engineering" (PNSE'97), Hamburg, September 1997, **Describes a non-sequential temporal logic for Petri-nets.**

[Lak94] C. Lakos, *Definition and Relationship to Coloured Nets*, Tech. report, Department of Computer Science, University of Tasmania, 1994.

[Lam77] Leslie Lamport, *Time, clocks, and the ordering of events in a distributed system.*, Communications of the ACM **21** (1977), no. 7, 558–656.

[Lea00] Doug Lea, *Concurrent Programming in Java (2nd edition) Design Principles and Patterns*, ISBN: 0-201-31009-0, The Java Series, Addison Wesley, 2000, **The book on concurrent programming in Java. Covers a lot of concurrency problems, discusses a lot of standard solutions. Does not cover formal techniques and it seems like everything is reinvented with the start of Java.**

[LF93] L.Bull and T. Fogarty, *Co-evolving Communicating Classifier Systems for Tracking*, Proc. Int'l Conference Neural Networks and Genetic Algorithms (1993).

[LLPY97] K. Larsen, F. Larson, P. Pettersson, and W. Yi, *Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction*, Presented at 18th IEEE Real-Time Systems Symposium. San Francisca, California, USA, December 1997, **Tries to solve the memory-requirements of reachability checks of timed automata**, pp. 14–24.

[LSMD96] Carine Lucas, Patrick Steyaert, Kim Mens, and Theo D'Hondt, *Reuse Contract: Managing the Evolution of Reusable Assets*, ACM Press, October 1996, **Describes how interface behaviour can be specified in an abstract way, aiming at reuse of the described interface.**

[Lyn96] Nancy A. Lynch, *Distributed Algorithms*, ISBN: 1-55860-348-4, Morgan Kaufman, 1996.

[MA97] John O. Moody and Panos J. Antsaklis, *Deadlock avoidance in petri-nets with uncontrollable transitions*, Tech. report, Technical Report of the ISIS Group at the University of Notre Dame ISIS-97-016, October 1997.

[Mil99] Robin Milner, *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, May 1999, **Extensive formal explanation of the π calculus. The concept 'mobility through channelling' is quite interesting.**

[Mon93] David J. Montana, *Strongly Typed Genetic Programming*, Tech. Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993.

[Moo96] Simon W. Moore, *Data-flow machines*, University of Cambridge, http://www.cl.cam.ac.uk/users/swm11/dataflowlecture/dataflowlecture.html #compare, 25 April 1996, **Historically interesting pages. It is surprising to see how the evolution of dataflow machines follows closely the evolution of Petri nets.**

[Mor96] L. Morgenstern, *The Problem with Solutions to the Framing Problem*, The Robot's Dilemma Revisited: The Frame Problem in Artificial Intelligence (K. M. Ford and Z. Pylyshyn, eds.), Ablex Publishing Co., Norwood, New Jersey, 1996.

[MSBW94] Arthur McCord Majority S. Blumenthal, Herbet S. Lin and Leslie M. Wade, *Academic Careers for Experimental Computer Scientists and Engineers*, ISBN:0-309-04931-8, Computer Science and Telecommunications Board, 1994, **Investigates how computer science contributions should be measured**.

[Mur89] T. Murata, *Petri nets: Properties, analysis and applications*, 1989.

[MW99] Ronaldo Menezes and Alan Wood, *Distributed Garbage Collection of Tuple Space in Open Linda Coordination System*, Proc. of the 14th International Syposium onComputer and Information Sciences (Kusadasi, Turkey), 1999, pp. 957–965.

[Nas50a] J.F. Nash, *The bargaining problem*, Econometrica **18** (1950), 155–162.

[Nas50b] ______, *Equilibrium points in n-person games*, Proceedgins of the National Academy of Sciences of the United States of America **36** (1950), 48–49.

[Nas51] ______, *Non-cooperative games*, Annals of Mathematics **54** (1951), 286–295.

[Nas53] ______, *Two person cooperative games*, Econometrica **21** (1953), 128–140, **The historical papers that gave impulse to a whole new field of research.**

[NU97] Marian H. Nodine and Amy Unruh, *Faciliating open communication in agent systems: The infosleuth infrastructure*, Agent Theories, Architectures, and Languages, 1997, pp. 281–295.

[OFM97] A. Olsen, O. Fgremand, and B. MllerPedersen, *System Engineering using SDL-92*, Norh Holland, 1997, **A Book an SDL92 and how to use it in a system engineering process**.

[OHE96] Robert Orfari, Dan Harkey, and Jeri Edwards, *The Essential Distributed Objects Survival Guide*, ISBN: 0-471-12993-3, John Willey and Sons, 1996, **A funny book, which covers CORBA as well as COM/OLE.**

[Ore98] Peyman Oreizy, *Decentralized software evolution*, Proceedings of the International Conference on the Principles of Software Evolution (IWPSE1) (1998).

[Pet62] C.A. Petri, *Comunication by Means of Automata (Kommunikation mit Automaten)*, **The original Petri-nets paper**.

[Pet81] James L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice–Hall, Englewood Cliffs, New Jersey, April 1981, **Chapter 7 describes how a number of extensions to Petri-nets break certain decidability issues.**

[PM93] P.Altenbernd and R. Milczewski, *Description of Timing Problems Using Petri Nets for Level-Independent Timing Verification*, Int'l Proceeding ACM/SIGDA Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU) (1993), **Describes a number of complementary techniques to verify the dynamic and static aspects of timed petri nets.**

[Pnu77] A. Pnueli, *The temporal logic of Programs*, 8th annual symposium on the Foundations of Computer Science, IEEE Computer Society Press, November 1977, **Introduces the idea of using temporal logic as the logic basis for proving correctness properties of concurrent programs.**

[PW94] Jing Peng and Ronald J. Williams, *Incremental Multi-Step Q-learning*, International Conference on Machine Learning, 1994, pp. 226–232.

[RB02] Michiel Ronsse and Koen De Bosschere, *Non-Intrusive Detection of Synchronisation Errors Using Execution Replay*, vol. 9, Automated Software Engineering, no. 1, Kluwer Academic Publishers, 1 2002, pp. 95–121.

[Reu] Ralf. H. Reussner, *An Enhanced Model for Component Interfaces to Support Automatic and Dynamic Adaptation*, Tech. report, Chair Informatics for Engineering and Science, Universität Karlsruhe.

[SAG98] Richard S. Sutton and Andrew G.Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, A Bradford Book, 1998, **A very good (formal) introduction to reinforcement learning. Q-Learning, SARSA and TD-learning are explained.**

[SB92] Nicol N. Schraudolph and Richard K. Belew, *Dynamic Parameter Encoding for Genetic Algorithms*, Machine Learning **9** (1992), 9–21.

[Sch87] Craig G. Schaeffer, *The ARGOT Strategy: Adaptive Representation Genetic Optimiser Technique*, Proceedings of the second international conference on Genetic Algorithms and their Applications (John C. Grefenstette, ed.), Cambridge, Lawrence Erlbaum Associates, July 1987, pp. 50–58.

[SEE99] *The SEESCOA Project Proposal*, http://www.cs.kuleuven.ac.be/cwis/research /distrinet/projects/SEESCOA, October 1999, **Home of the SEESCOA research project.**

[Sel01] Bran Selic, *The Unbearable Lightness of Distributed Systems: a Survivors Guide*, Keynote Speech at the Components for Mobile Computing workshop, 38th International TOOLS Conference, Zürich, Switzerland (2001), **Explains why programming open distributed systems is very difficult.**

[SGP94] Bran Selic, G. Gullekson, and P.T.Ward, *Real-Time Object Oriented Modelling*, John Wiley and Sons, Inc, 1994, **The book about ROOM**.

[SJ75] G. J. Sussman and G. L. Steele Jr., *Scheme, an Interpreter for Extended Lambda Calculus*, **The original paper that introduces Scheme. The paper itself is written on a good old fashioned typewriter, and some nostalgy can be found between the lines. A must read.**

[SKML] Luc Steels, F. Kaplan, A. McIntyre, and Joris Van Looveren, *Crucial Factors in the Origins of Word-meaning*, Tech. report, Sony Computer Science Laboratory - Paris, VUB Artificial Intelligence Laboratory - Brussels.

[Smi] S. F. Smith, *A Learning System Based on Genetic Adaptive Algorithms, Doctoral thesis*, Department of Computer Science, University of Pittsburgh, Pittsburgh 1980.

[SS89] Dale Schuurmans and Jonathan Schaeffer, *Representational Difficulties with Classifier Systems*, Proceedings of the Third International Conference on Genetic Algorithms (San Mateo, CA) (James D. Schaffer, ed.), Morgan Kaufman, 1989, pp. 328–333.

[Sug98] Y. Sugiyama, *Runtime software evolution based on version management*, 1998.

[Sut88] Richard S. Sutton, *Learning to Predict by the Methods of Temporal Differences*, Machine Learning **3** (1988), 9–44.

[Tes92] Gerald Tesauro, *Practical Issues in Temporal Difference Learning*, Advances in Neural Information Processing Systems, vol. 4, Morgan Kaufman Publishers, Inc., 1992, pp. 259–266.

[TM87] Tommaso Toffoli and Norman Margolus, *Cellular Automata Machines, a New Environment for Modelling*, The MIT Press, Cambridge Massachusetts, 1987.

[VHT00] A. Vallecillo, J. Hernandez, and J. Troya, *Component interoperability*, Tech. Report ITI-2000-37, Departemento de Lenguajes y Ciencias de la Computacion, University of Malaga, July 2000, **Available at http://www.lcc.uma.ex/ãv/Publicaciones/00/Interoperability.pdf. This paper gives a good overview of the problems involved withou component/object interoperability. It starts with basic syntactical conflicts and end with semantic interoperability.**

[Vre95] Gerard A.W. Vreeswijk, *Formalising Nomic: Working on a Theory of Communication with Modifiable Rules of Procedure*, Tech. report, Technical Report CS 95-02, Department of Computer Science, University of Limburg, Maastricht, The Netherlands, 1995.

[WMC01] Bernhard Westfechtel, Bjorn P. Munch, and Reidar Conradi, *A layered architecture for uniform version management*, IEEE Transactions on Software Engineering **27** (2001), no. 12, 1111–1133.

[Wyd01] Bart Wydaeghe, *PacoSuite: Component Composition Based on Composition Patterns and Usage Scenarios*, Ph.D. thesis, Vrije Universiteit Brussel (VUB), November 2001.

Index