Towards a Minimal Object-Oriented Language for Distributed and Concurrent Programming [1]

Matthias Radestock and Susan Eisenbach
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
Email: M.Radestock,S.Eisenbach@ic.ac.uk

July 28, 1994

[1] The research has been financially supported by the DTI, grant ref: IED4/410/36/002.

Abstract

The aim of the project was to develop and implement a concurrent object-oriented language for distributed programming. Based on an analysis of the state of the art in the design of object-oriented systems and an abstract but very simple and compact definition of the object-oriented paradigm, we first develop a model for objects that is simple and incorporates only essential features. It is shown how inter-object and intra-object concurrency arise naturally from this model. Taking a look at the problems which arise when implementing a concurrent and distributed object-oriented system, this model is then refined and modified. A formal specification of the final model - the basis for the implementation of a system - is presented.

The second part of the project was devoted to the design of a language and the precise definition of its syntax and semantics. The language has only a few constructs. This minimalistic approach makes it easier to reason about it. High-level object-oriented features are not included as primitives but it is shown how these can be modelled using the existing constructs. As concurrency issues were expected to play an central role it was important to define the semantics in a way that enables reasoning about these issues. It was decided to use Milner's π-calculus for that purpose. This calculus is well-defined and provides a convenient way of modelling concurrency. As the intention was to design a minimal language it seemed appropriate to describe its semantics in terms of a calculus which itself only has a minimal set of constructs. However, it was found that the calculus lacked the expressiveness needed to define the semantics of the language. Therefore we introduce a new operator, called *mismatch*, into the calculus and investigate its impact on the rest of the calculus.

Finally we describe the implementation of two different systems that demonstrate the correctness and usefulness of the design of the object model and the language. In the first approach the language is translated into π-calculus which is then executed using a π-calculus interpreter. The translation is also useful for investigating behavioural characteristics and properties of programs by analysing them in their π-calculus representation. Formal methods that are applicable to the calculus can be employed. The second approach implements a system based on the developed object model. This implementation is carried out using Darwin and Regis - a configuration language and run-time system for distributed applications. By translating the language into Regis it thus becomes executable on a distributed system.

Contents

List of Figures

Chapter 1

Background

This chapter contains an analysis of some of the current research in the field of object-oriented systems. It provides the background and the motivation for the project. The research community in this area is very large, the topics of research are very diverse. There are a multitude of different views. Hence the following analysis is bound to be incomplete and to a large extent is centred around the author's opinion.

The information provided in this chapter is the *foundation* for the work. Background information that is merely needed to *understand* the content is provided in the appendix and referred to from the context where it is needed (e.g. an introduction to the π-calculus and Darwin/Regis). Also the relevance of some background information (e.g. a discussion of the inheritance anomaly) becomes only clear at the design stage in which case the information is provided there rather than in this chapter.

The project uses knowledge from a wide area in the field of computer science. We presuppose some familiarity of the reader with the object-oriented paradigm, including mechanisms like inheritance. Further areas are concurrent computation, distributed systems, compiler theory, model theory, language theory, semantics and formal methods in general.

1.1 Introduction

The era of Object-Oriented Design (OOD)[CY91, Boo94, R+91] and Object-Oriented Programming (OOP)[Cox86, Mey88] started in the early eighties and since then these areas have become increasingly popular. The appeal of the object-oriented paradigm lies in its apparent simplicity. However, a great variety of both object-oriented design methods and object-oriented programming languages have emerged and there are many different views of what the object-oriented paradigm actually is. In the beginning it was still possible to convert models and programs from one design/programming approach to another without great difficulty. But when attempts were made to apply OOD and OOP in new areas it became increasingly obvious that the lack of a unified paradigm led to major differences between the approaches. One of these areas is concurrent programming. Many models have been developed to apply OOD in this area. They approach the problem in several different ways. The lack of a precisely defined paradigm led to false claims about the object-oriented-ness of many of them. Often we find that the object-oriented paradigm is simply redefined in a way that makes it more suitable for the particular approach. The failure to adequately represent some of the essential features of the paradigm is thus disguised. In our approach we shall therefore precisely define what our understanding of the paradigm is and justify that our definition indeed captures *all* and *only* the essential features.

The major motivation for designing concurrent object-oriented languages is the advantage that is achieved by unifying the abstraction for processor and memory (cf. [KL93]). The object-oriented paradigm provides a means for specifying composite data structures and associating operations with those structures. Thus it incorporates all aspects of the *abstract data type* paradigm while

at the same time putting the emphasis on the functional rather than the structural part of the definition. Consequently it includes a variety of mechanisms related to the functional part, e.g. inheritance, defaults, polymorphism. The paradigm will be discussed in more detail in another section of this paper. The process paradigm allows us apply a decompositional approach for specifying functionality. A process describes the interaction between its sub-processes, i.e. their communication, synchronisation, creation and destruction. Thus the overall functionality of a process is determined by the functionality of its sub-processes and the specified interactions. Hence, while the object-oriented paradigm is a model for data abstraction, processes are models for control abstraction. The concurrent object-oriented paradigm combines both by associating processes with objects. This will also be discussed in more detail at a later stage.

1.2 Objects

The basis for this paper is a definition of the term *object* which is in some sense minimal. In the fundamental panel discussion about the term in [W+91] it was pointed out that the notions of *classes* and *inheritance* are *not* essential features of the object model. Our approach is to aim for a definition whose key features are simplicity, clarity and completeness. It should also be noted that it is an implementation based rather than a design based definition. By taking into account issues like efficiency we lay the foundations for an implementation of an interpreter for a language based on the model.

The definition of the term *object* is shown in Figure 1.1. This definition is a combination of

Something is an object if and only if it:

- has a state
- can receive messages from objects
- may change its state due to the arrival of a message
- can send messages to other objects due to the arrival of a message

Figure 1.1: Definition of the Term *Object*

static (state) and dynamic (messages) aspects and the interaction between them.[1] Concurrency deals with dynamic aspects. The definitions of both *intra-object concurrency* and *inter-object concurrency* are based on the idea of *message passing*. The difference between both is founded on the notion of *boundaries* for objects. The boundary of an object is clearly defined by the definition of its state. The state information of an object is encapsulated by the boundary. Normally only the object itself has access to it. Access to other object's state information is only possible via message transfer and it is up to the receiver whether and how the state information is made accessible. Hence the above definitions allows objects to exist and act concurrently - each operating on its own state and communicating by exchanging messages:

Intra-object concurrency The concurrent reception and interpretation of messages within one particular object.

Inter-object concurrency The interpretation of arriving messages by several objects concurrently.

Thus both intra-object and inter-object concurrency arise naturally from the definition of the term 'object' and hence can be viewed as intrinsic features of the object-oriented paradigm and systems based on it.

[1] The notions of *state* and *messages* will be further investigated at a later stage.

1.3 Recent Approaches

We investigated existing models of concurrent computation in general [Neu91, Mil89, BA90, Sch86] and of object-oriented models [W+91, Atk91] and carried out a survey on existing concurrent object-oriented languages, similar to the one presented in [KL93]. We found that most attempts to facilitate the object-oriented paradigm for concurrent computation extend an existing (sequential) object-oriented language. Unfortunately most of these languages, with some notable exceptions (e.g. *SmallTalk* [GR89]), are already based on a much too narrow definition of the object-oriented paradigm. The extensions to allow concurrent computation not only inherit these shortcomings but tend to emphasise them, because they rely exactly on those features of the paradigm which have been neglected. Some of the languages only allow for *inter-object concurrency* while others also incorporate *intra-object concurrency*. There exist a great variety of synchronisation techniques. In many approaches (e.g. guarded methods) the introduction of concurrency is in conflict with the concept of inheritance - an observation known as the *inheritance anomaly*[Fro92, JA92, KL93, BY87, YM93]. We will discuss this in more detail when dealing with inheritance in our model. Common to all these extended languages is that one can *see* that the concurrency features were only added at a later stage. As a result programming in those languages is less intuitive. Flexibility and universality are limited.

Other approaches are based on the definition of entirely new object-models. Examples are the notion of *actors* and the concept of *group-reflective architectures*. Both manage to deal with concurrency without adding special synchronisation methods. Unlike the extensions of existing object-oriented languages the concurrency arises naturally.

1.3.1 Actors

In the actor model objects [Agh86, YT87] are associated with message queues. Actors communicate with each-other by appending messages to the queue of the receiving actor. Actors remove messages from their queue and, depending on which message was received, perform some actions. The mapping from messages to actions is specified by a behaviour script. After the actions associated with a message have been performed, the actor becomes a new object, i.e. the next message in the queue is parsed by an actor with a potentially different behaviour script. The inter-object concurrency arises naturally as all the actors in the system may parse their message queues concurrently. Intra-object concurrency we get when an actor specifies its new behaviour before it has performed the last action associated with the received message. In that case the new object starts processing the next message in the queue while the original actor is still dealing with the previous message. A detailed discussion of the actor model is beyond the scope of this paper and we refer the interested reader to the literature on that subject.

There is no explicit notion of values and state. Thus it is not clear how inheritance can be incorporated into the model. The actor model is a very elegant approach to the problems of concurrency. However, because of the missing notion of state in the way it is commonly understood (i.e. a set of attributes with associated values) and the absence of a straightforward simulation of this aspect of the object-model, we can't call the actor model an object-oriented model. This is probably the reason why the focus of attention within the research community in object-oriented programming has shifted away from actors in recent years. Actors are not dead though, for the simplicity of the approach is really striking.

1.3.2 Group-Reflective Architectures

Unlike actors the group-reflective architectures [MWY91] are based on high-level object-oriented features like inheritance and the notion of meta-objects. As a result the patterns of synchronisation and interaction in general are very complicated. The design and implementation of a system based on this approach might prove to be difficult. As the model itself is already rather complex the object-oriented design might lack the simplicity it had before. This view is further supported by the fact that a completely new notion is introduced on all levels from design to implementation

- the notion of *groups* (hence the name). It is still to early to say what can be achieved with group-reflective architectures. Only few results have been published so far.

Incorporating high-level features into the model has its advantages, mainly that the implementation of these features can be optimised and the resulting system is efficient in their execution. The disadvantage is that once these features have been specified in the model, modifications to them become impossible. This wouldn't be that problematic if there was clarity about the exact definition and specification of the features, but in the research area of object-oriented systems the meaning of many of these features (like the notion of inheritance) has not been determined sufficiently exhaustive. Furthermore, imposing these features and new ones (i.e. groups) on the user's design and implementation makes both of these processes more difficult in cases where the features are not needed. Consequently the resulting implementation also executes less efficiently.

1.4 Object-Oriented Systems

The definition of objects includes the notion of *state*. It is this notion that causes the difficulties one faces when trying to model the concurrency aspects. The state of an object affects its behaviour. As the state information of objects is hidden from the outside observer the observable behaviour of an object comprises the messages it sends to other objects when processing an incoming message. How an object reacts to the arrival of a message at a specific moment in time is entirely determined by its state and this state may change due to the interpretation of messages. Hence the order of interpretation of messages is critical and this both at inter-object and intra-object level.

1.4.1 Interactive Systems

The state of the system is a snapshot of the states of all its objects.[2] At specific times we want this global state to be independent from any non-deterministic choices made during the course of computation, i.e. the states we get from different computational paths must be the same. Typically we would like this to be the case at the end of the computation. The global synchronisation at specific times can only be achieved if we introduce some mechanism to the system that guarantees the same order of message interpretation for every object in the system. This does not mean, that the order of interpretation must be the same system-wide, we only have to ensure that each particular object interprets the messages in the same order so that the sequence of states that object is going through is fixed (unless we explicitly wish a non-deterministic behaviour). The global synchronisation can then take place at the time where no messages are left to interpret for any object. There is an analogy to the *confluence* property of functional languages which itself is a result of the Church-Rosser theorem. The theorem states that the result of an evaluation of a term is independent from the order in which the sub-terms are evaluated. Parallel evaluation of programs written in a functional language relies on this property. A similar observation in the object model is the basis for inter-object concurrency, i.e. provided that each individual object goes through the same sequence of states the final state will be the same.

Since most object-oriented systems are interactive, we need to define what *deterministic evaluation* means for them. Instead of just one state which is of interest to the outside observer (the final state) an interactive system has a number of such states. They are reached when some external action is required. The system is then in a state where it waits for a message to be sent to an object. This message has to come from an external source. The system then proceeds until it reaches such a state again. It is also possible to divide the system into parts which are active or inactive. The states of inactive parts can be compared. This enables us to compare the state of an interactive system where messages are still being interpreted when we restrict the comparison to the inactive parts (which wait for an external action). However, a detailed investigation of interactive systems in this context would be beyond the scope of this paper.

[2] Note that obtaining such a global system snapshot is a non-trivial task in a concurrent and distributed system.

1.4.2 Sequential Object-Oriented Systems

The reason why all early object-oriented systems abandoned both concurrency features and interpret exactly one message at a time system-wide, is that this technique guarantees the same order of message interpretation system-wide. While such an approach of course fulfils the requirement of a deterministic evaluation, this goal is achieved by "unnecessary" sequentialisation and weakening of the definition of the object-oriented paradigm (according to our definition of the term *object* the object itself should interpret the message).

Even a sequential object-oriented system is not without difficulties: An object may send a message to itself directly or indirectly via other objects (i.e. it sends a message to that object which in turn sends a message to the sender). This from the first look seems to imply that we need both types of concurrency in order to avoid deadlock: clearly this message has to be interpreted while the interpretation of the message in the object which sent it has not been finished yet! Thus if the sender and the receiver are not identical we need inter-object concurrency and otherwise intra-object concurrency. However, there is a mechanism to avoid this: During the interpretation of a message every time messages are sent to other objects or to the object itself, the message interpretation of this message must be interrupted and the interpretation of the message sent starts. When the interpretation finishes the interrupted interpretation continues. Thus the sending of a message and the finishing of an interpretation initiate a *context switch*. In such a system sending a message is equivalent to invoking a procedure in an imperative language.

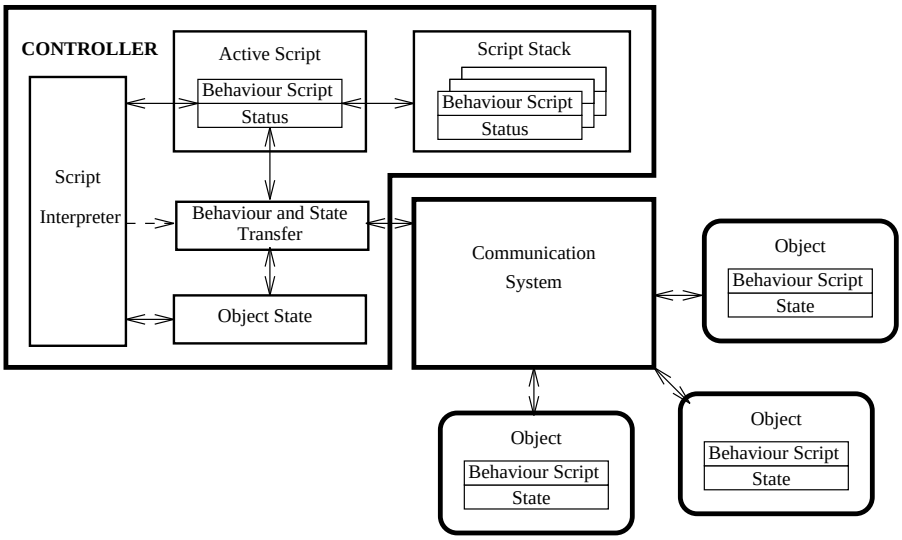

Figure 1.2: Conservative Object-Oriented System

The structure of such a conservative object-oriented system is shown in Figure 1.2. The controller is basically a processor operating on data which is provided by objects (i.e. their behaviour script and state). It has a stack for storing the behaviour scripts and states of objects whose message interpretation has been interrupted. It has registers which control its state and it executes instructions from the script language. The *behaviour script* of an object determines its behaviour, i.e. it specifies what happens when certain messages are received. We will investigate this issue in more detail at a later stage. The communication system transfers scripts and state information from and to the objects on request of the controller.

For the type of object-oriented systems we have looked at so far, the interpretation of messages takes place centrally which guarantees the system-wide exactly-one-message-at-a-time-interpretation. But we want the message interpretation to take place inside the object, because this is much closer to our definition of objects.[3] Hence we could construct a system where each object can interpret

[3] Note the difference between message-passing and message-interpretation we get in this kind of system.

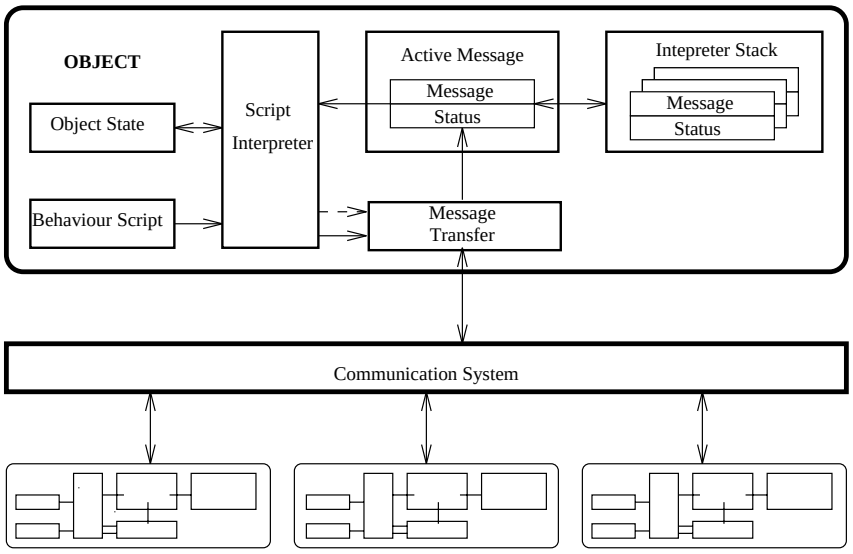

Figure 1.3: Distributed Message Interpreter

multiple messages with the restriction that the interpretation is interrupted when a message is sent and that we start with sending exactly one message to an object. However, while this is an approach that complies with the definition of objects much better than the centralised message-interpretation, it still does not provide any real concurrency. We have achieved a distributed message interpretation (Figure 1.3), though there is only one message interpretation active at any time. It might of course be the case that there is no interpretation active at all, which happens when we've reached the final state.

1.4.3 A Model for Concurrent Objects

The preceeding discussion revealed how the sequentialisation in an object-oriented system is achieved by special constraints and using special mechanisms. We therefore obtain our first model for concurrent objects by removing these constraints (Figure 1.4).

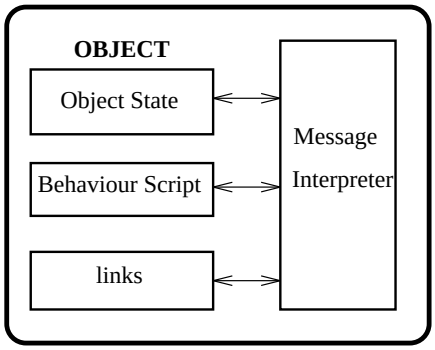

Figure 1.4: First Model for Concurrent Objects

An object consists of four parts: state, behaviour script, links, message interpreter. The message interpreter receives messages along the links from other objects and interprets the behaviour script and accesses the state accordingly. It also sends messages to other objects along the links. We've already generalised the model used in the previous pictures. Objects can now modify their behaviour script. This requires an interpreter-based language as the behaviour scripts can't be pre-compiled in that case. The message interpreter can have the the capability to interpret several

instances of the behaviour script simultaneously (thus enabling intra-object concurrency) or alternatively to use a context switching mechanism as described above. Care has to be taken to avoid conflicts when accessing the object state simultaneously from several threads of behaviour script interpretation. We will further investigate this issue at a later stage. Inter-object concurrency in this object model arises from the independence of the message interpreters of different objects.

1.5 Conclusions

We showed how concurrency arises naturally from a proper definition of the object-oriented paradigm. Despite this most concurrent object-oriented systems fail to represent the concurrency features in an equally natural way. Recent research shows some promising progress but an adequate solution still has to be found. An investigation of common types of object-oriented systems revealed how their design was affected by the attempt to *avoid* concurrency. Removing these constraints we obtained a first model for concurrent objects. It remains to be shown how this model can be refined in a way to overcome the problems one commonly faces in the area of concurrent object-oriented systems. This will be done in the following chapters.

Chapter 2

Design of an Object Model

The definition of the term *object* incorporates the notions of *state, message passing* and *behaviour*. Defining the meaning of these notions is the basis for developing an object model. Thereby we get an insight into the very nature of objects and object-oriented systems. Our aim is to find a simple model for describing these three categories. Thus the model which we develop in this chapter is in some sense minimal.

2.1 Towards an Object Model

In this section we carry out a step-by-step refinement of our object model. At each stage the remaining problems are discussed and the next stage is obtained by solving these problems. Thus in the end every aspect of the model can be justified as the solution to a problem that had occured at some stage during the design.

2.1.1 The State of an Object

The behaviour of an object is determined by its behaviour script. When the script is interpreted the object's state is accessed. Thus the current object state affects the behaviour of the object. How do we model this state? From the designer's point of view the state of an object is an assignment of values to a set of attributes. But this causes severe problems as it seems to enforce the separate modelling of attributes and values. The attributes have a type. Normally this type is specified in terms of a set of possible values, e.g.

attribute colour={red, green, blue}

This approach enforces the atomicity of values - we can't decompose them. Furthermore we need a separate category in our model for expressing values. The specification of the attribute types introduces yet another category and we need some means of defining sets. But sets are just an abstract data type and our intuition tells us that therefore they should be modelled as an object, i.e. we would model the type of an attribute as an object. However, we actually don't necessarily need the explicit notion of types in the first place. We can always model it with the means we already have when it is required - an approach taken in languages like *SmallTalk*[GR89]. Not assigning types to attributes, those can now potentially take any value.

We'd also rather like to express values in terms of objects. In our example we would then have three objects - red, green and blue. The set of attributes then becomes a set of object references (e.g. if the colour is green the attribute has a reference to the object green). Again *SmallTalk* is an example for such an approach. As we shall see later, we can even go a step further by uniting the set of attributes with another set of references - the set of parts. How we get this second set is described in the next section.

2.1.2 Object References and the Decomposition Hierarchy

In order to communicate with each other by sending messages objects must have references to other objects (the receiver of a message in particular). We have seen in the previous section that we also need object references for modelling the state of an object.

Generating Object Identifiers from Positions in Hierarchies

Theoretically each object in the system can be referenced by any other object or even by itself. Thus we need some system-wide *object identity*. One way of identifying objects is to assign an identifier to them at the time they are created. As this identifier has to be unique the creation of it has to take place centrally. For a distributed system this means a huge amount of communication taking place at a central location. Furthermore there is no way of referring to objects before they come into existence. This imposes major constraints on the design of any implementation based on such a model.[1]

The problems don't arise if the system is static - i.e. all the objects that are to be created are known in advance. Then we can assign object identifiers at compile time. Unfortunately, in a complex system at most the large grain objects are known in advance. If objects are used on all levels, dynamic creation of objects is an essential feature without which no useful computations can be carried out.

The second approach uses an important property of the object-oriented design method: objects are incorporated in two hierarchies - the abstraction hierarchy and the decomposition hierarchy (examples of both hierarchies are given in Figures B.1 and B.2). The former is not a tree structure if multiple inheritance is supported. The decomposition hierarchy, however, is always a directed tree. This property makes it more suitable for our purpose - a fact that has been recognised in recent research (cf. [PB94]).

Sorts

OID

Constants

$\texttt{Universe} : \text{OID}$

Predicates

$\mathsf{contains}(\text{OID}, \text{OID})$

Axioms

$$\mathsf{contains}(x, y) \longrightarrow x \neq y \tag{2.1}$$

$$\mathsf{contains}(x_1, y) \wedge \mathsf{contains}(x_2, y) \longrightarrow x_1 = x_2 \tag{2.2}$$

$$\neg\mathsf{contains}(x, \texttt{Universe}) \tag{2.3}$$

$$(\exists x.\mathsf{contains}(x, y)) \vee y = \texttt{Universe} \tag{2.4}$$

Figure 2.1: The contains Relation

Figure 2.1 describes the decomposition relation in terms of an axiom system. The relation is defined over *object identifiers*(OID). Thus the names of all graph nodes have to be unique. Axiom (2.2) ensures that any node has at most one predecessor. The top node in the hierarchy is called `Universe`. Hence it doesn't have a predecessor and is unique (i.e. it's the only node which doesn't have a predecessor) (axioms (2.3) and (2.4)). The implicit for-all-quantification in axiom (2.4) also guarantees that the graph is fully connected and includes all elements of the domain.

[1] There are solutions to overcome these problems. However, they all involve a considerable overhead.

A different representation (Figure 2.2) doesn't require the global identifiers property. Pro-

Sorts

SEQ_{NAME}, NAME

Constants

$\texttt{Universe}$: NAME

Predicates

$\texttt{path}(\text{SEQ}_{\text{NAME}})$

Axioms

$$\texttt{path}(s < x >) \longrightarrow \texttt{path}(s) \tag{2.1}$$
$$\texttt{path}(< x > s) \longrightarrow x = \texttt{Universe} \tag{2.2}$$

Figure 2.2: Using *Paths* as Identifiers

vided that unique names are assigned to the immediate successors of a graph node we can devise a function which returns a global unique name for any node in the graph. This could be done by simply taking the path from the root of the tree to the node. We can't describe this representation in terms of a relation. What we have to do instead is to specify an abstract data structure for this type of graph. Then we can show that the set of unique identifiers that are returned by the function, and the relation between these identifiers as obtained from the graph, obey the axioms defined in Figure 2.1. As the *path* relation is unary its models can be represented as a set (of sequences of names). It turns out that this set can be mapped to the set of *object identifiers* from Figure 2.1. We can also get an appropriate mapping for the constants and predicates:

$$\texttt{path}(s) \longleftrightarrow s \in \text{OID}$$
$$\texttt{path}(s < x >< y >) \longleftrightarrow \texttt{contains}(s < x >, s < x >< y >)$$

For a more detailed discussion of these issues see [RS92]. In [BC87] the importance of the decomposition hierarchy is stressed. However the approach presented there implements these *part hierarchies* (as they are called there) on top of the abstraction hierarchy. We make it the *basis* of our approach.

Communication in a Hierarchical Model

One of the aims of object-oriented design in hierarchical systems is to reduce communication between objects in the hierarchies to communication between neighbouring nodes. Thus the hierarchies provide a means for the design of a distributed system - the nodes can be distributed across a network of computers and the needed connections are known in advance. By using path names as described above the unique identifiers can also be used to locate the object.

As the graph is fully connected point to point communication (i.e. communication between any two objects) can be modelled wholly in terms of communication between neighbouring nodes (Figure 2.3). But this is not always a desired feature as it concentrates communication in the top nodes in the graph. For instance in a binary decomposition hierarchy 50 percent of the point to point communication would be channelled through the top object.[2] If the physical configuration of the distributed system matches the decomposition hierarchy, i.e. the system itself is a tree structure, this is the most efficient way. However most distributed systems are either flat or their hierarchy doesn't match the decomposition hierarchy. In case of a flat structure this would mean

[2] Assuming that on average any point-to-point communication is equally likely.

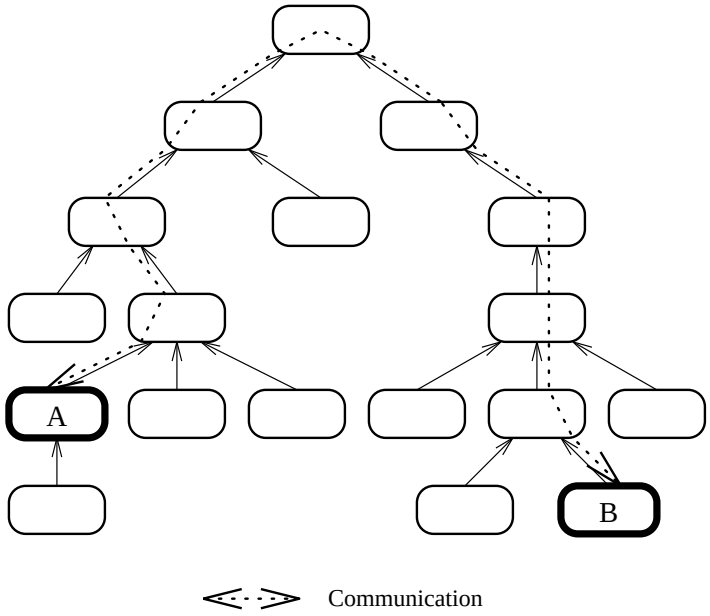

Figure 2.3: Communication Between Objects

that although a point to point connection between two objects is possible the information flow is channelled through many objects. It seems therefore reasonable to use the described method only for the connection setup. If an object wishes to send a message to another object it finds its location with the method described above. Then a private communication channel between these two objects is set up which is independent from the hierarchy. Figure 2.4 gives an example of the connection setup and the actual communication between two objects in such a system.

The problem of concentration of communication activities in the top nodes of the hierarchy, actually appears to be less serious when we deal with the decomposition hierarchy of an object-oriented system. Objects mostly communicate with their sub-objects, container objects and fellow sub-objects and references to these are thus logically short links. However, modelling data types as objects results in a high number of references to the objects representing the data types. These objects can usually be found in a library that forms a part of the decomposition hierarchy. Thus the references from inside the part of the decomposition hierarchy that represents the actual application program correspond to logically long links.

As the previous discussion reveals, the decomposition hierarchy is the basis for a convenient way of modelling communication. But if the hierarchy is used for that purpose it necessarily has to be built into the system. Hence the most abstract view, that an object-oriented system is a set of objects with a set of (binary) relations, has to be specialised. As the decomposition hierarchy is used for generating object identifiers, all other relations between objects are now modelled using this relation. This together with the preceding discussion of the model for the object's state leads us to our first improvement of the model (Figure 2.5). Note that *value* in this picture is an *object reference* (i.e. a path) as opposed to a *link*. Each object is assigned a *set of parts*. This is a set of pairs *(object name, hard link)*. *Hard links* are links of the underlying communication system. The set of parts of an object includes links to exactly those objects which are part of it. An additional hard link exists to the container object (the object it is part of). Using these links all the communication as described above can take place. Note that, although we used the term *hard* for the description, the links are set up and destroyed dynamically during the evolution of the system. The link to the container object can't be changed. Hence we don't allow objects to move around in the decomposition hierarchy. This seems to be an unnecessary restriction, but without it a complicated procedure of updating references would become part of the model. The same holds for the names of the parts. They too can't be changed although this restriction is not

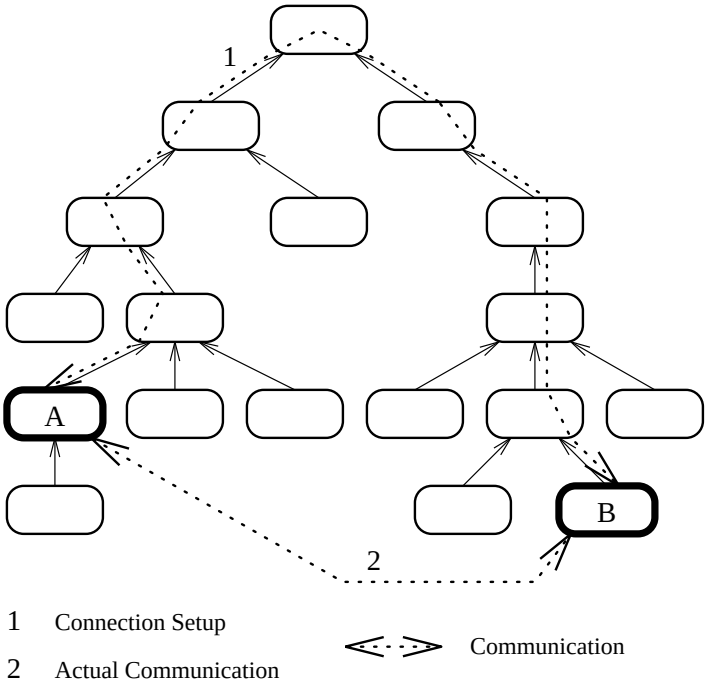

Figure 2.4: Connection Setup

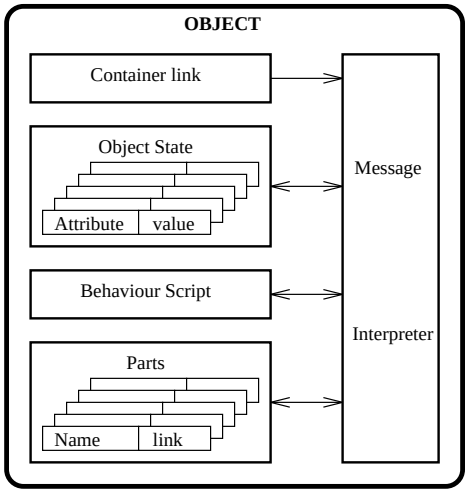

Figure 2.5: Improved Object Model

expressed in the figure. However, dynamically creating and removing parts is possible.

2.1.3 Unifying the Set of Parts with the Set of Attributes

As we have mentioned earlier we now go even further in simplifying the model. The entries in the set of attributes are ordered pairs *(attribute name, object reference)*, provided we use references to model attribute values as it was discussed in the earlier section. The entries in the set of parts are ordered pairs *(object name, hard link)* too. What we now try to achieve is the representation of the set of attributes in terms of the set of parts. First we introduce another element in the model of objects: *value*, which is of the fixed type *object reference*. An attribute can now be represented as a part of the object with the name derived from the *attribute name* and the associated object reference stored as the *value* of the attribute object. Thus we get a further improved model for objects (Figure 2.6).

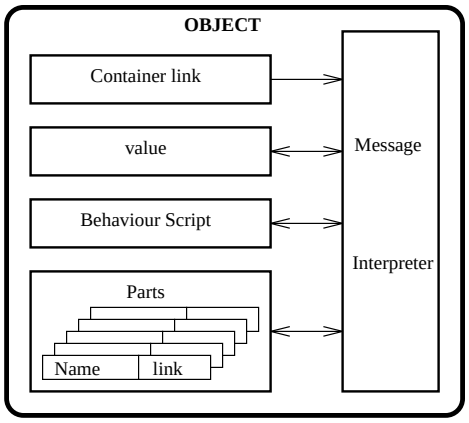

Figure 2.6: Further Improvements to the Object Model

Although this modification to the model seems simple and straightforward its implications are far reaching. There is no longer a distinction between the parts of an object and its attributes! Hence there are no special operations defined for attributes.

The model provides an abstract view of communication in an object-oriented system. It turns out that the following things happen in such a system:

- objects establish connections to other objects.
- objects send messages along an established connection and receive values as replies.
- objects initiate the creation or destruction of parts of other objects.

These are the only activities which take place in a system based on this model. It remains to be shown how these operations relate to the activities commonly found in object-oriented systems. This will be discussed in the next section.

2.1.4 Behaviour Description and Messages

We now have the means for modelling the state of objects and the structure of an object-oriented system and its underlying communication system, and wish to use this for modelling the behaviour of objects. Thereby we have to take into consideration two different aspects:

1. the exchange of messages
2. the actual behaviour description

While the first aspect deals with the message transfer in general, the second focusses on the creation and acceptance of messages, in other words: which messages cause which reaction.[3]

Message Passing

The most abstract format of a message is

$$result = object.message(arg_1, arg_2, \ldots, arg_n)$$

where $result, object, arg1 \ldots argn$ are all references and *message* is the name of a message. Statements of this form occurs in the behaviour description of an object. When it is executed the semantics of this statement is the following:

1. A connection from the current object to *object* is established.
2. The message and the arguments are transferred along the channel.
3. A result is received along the channel and assigned as a value to the object referred to by *result*.

Some implementations allow the sender of a message to continue with the interpretation of its script as long as the result is not needed, thus adding real concurrency. But this concept of so-called *futures* [WFN90] destroys the property of determinism if it is applied in general. The receiver of the message might send (directly or indirectly) messages to other objects which are also being accessed by the sender. So potentially messages could arrive at other objects in an arbitrary order. Furthermore we experience difficulties when we try to incorporate this into our model where *result* is a reference to another object. While it is easy to determine when this object is accessed again in the current behaviour script, and hence to wait until it actually is updated, it is nearly impossible to do this for all objects. But this is required as in our model other objects might access the *result* object as well. So in fact we'd have to tell all objects (apart from the object the message was sent to) to wait for an update of the *result* object if they wish to access it. This is impracticable to implement in a distributed system.

We need a second statement - one for returning a result

return *result*

where *result* is a reference. The value is transmitted back to the sender of the message. The *return* statement must occur exactly once as it synchronises sender and receiver.

Script Language

When a message is received by an object, statements in the behaviour description are executed. But how can the script of the behaviour description determine the course of the computation? One could define a complete language for the behaviour script. This language would have to incorporate several statements:

- a *case* statement for selecting different parts of the behaviour script for execution depending on the received message and the state of the object.[4]
- test of references for equality - this is needed to express the dependency from object's state in the *case* statement. The state is represented by the values of the sub-objects. These values are references.
- the two message passing statements from above (sending a message and returning a reply)
- creation and destruction of objects.

[3] A reaction is the sending of messages to an objects.

[4] In our model this is the values of the objects which are part of the current object.

- test whether an object exists - this is needed to express the dependency from the system state in general.
- creation and destruction of local variables - we need this to store the arguments of a received message.

In order to dynamically create objects we have to be able to assign behaviour descriptions to objects dynamically. Further problems arise because objects can refuse messages. Therefore a whole set of statements for error handling has to be added. And the notion of dynamic local variables is something which doesn't match very well with our model where we have already eliminated the notion of state variables. It would thus require the introduction of another category into our object model and we would need special operations to access local variables and assign references to them.

In the second approach the notion of local variables is not needed. It's based on the following idea:

- for every message that can be received a object is created which is part of the current object and has the name of the message.
- for every local variable of a message that can be received a object is created which is part of the message object (see previous point) and has the name of the local variable.
- each object has an associated behaviour script.

So far it doesn't appear to be very different from the previous solution, but rather surprisingly it turns out that we can now abandon all statements of the behaviour script dealing with the message receipt (e.g. the *case* statement and the notion of local variables). What remains is:

- statements for creation and destruction of objects.
- a statement for initiating the interpretation of an object's behaviour script.
- a statement to return the result of the interpretation of the behaviour script.

The approach is similar to object models which treat the receipt of messages as execution of *methods*. The difference is that our model treats methods as objects and not even as special but ordinary objects. Any object can be seen as a method which returns a value. Further simplifying the model for objects we replace the *value* with the *behaviour*, as a basic value can be modelled as a method which does nothing but returns this value.

2.2 A Model for Objects

The minimalistic approach we've taken results in a definition whose only key elements are *links, object references* and *behaviour scripts*.

- Objects exist and act concurrently.
- Objects are nodes in a decomposition hierarchy.
- An object has a link to its container object in the hierarchy and to its sub-objects. All communication with other objects takes place along these links.
- Objects are addressed by *references*.
- References are lists of names, denoting paths in the decomposition hierarchy. The usual operations for lists apply, including test for equality.
- Objects can be created or destroyed.

- The existence of an object can be tested.
- An object has an associated *behaviour script*.
- The behaviour script of an object can be set, retrieved or evaluated.
- The evaluation of a behaviour script yields a reference.
- References can be converted to behaviour scripts.

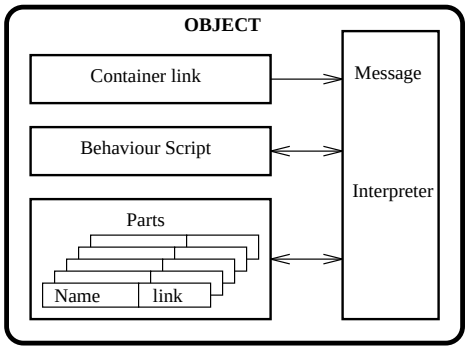

Figure 2.7: Final Object Model

The internal structure of objects based on this model is shown in Figure 2.7.

2.2.1 The Formal Specification

An axiom system for the object model is presented in Figures 2.8 and 2.9. The scheme used is a variant of an object-oriented specification language introduced in [Rya93]. The first line of the specification says that objects in this model are referred to by *links*. It also implicitly declares a global predicate object(LINK). The predicate is true if and only if an object with LINK as an identifier has been created.

There are six constant functions defined. interpret interprets a behaviour script in the context of the object. The context is provided in form of a reference to the object. The evaluation results in a path. toBehaviour converts a path into a behaviour - that is a behaviour script which returns exactly this path when interpreted. The auxiliary predicates newObj and deleteObj are provided for creating and destroying objects. getPath finds the shortest connection between two objects in the hierarchy. If the first two arguments are not global references the second argument is returned as a result thus making no changes. Otherwise a relative path is created from the object referred to by the first path to the object referred to by the second path. This is done by first removing the common prefix and then adding a suitable number of "ups" at the front of the remaining second path, thus creating a relative path from the first object to the second object. The latter operation is performed by the predicate shift. An example for the predicate would be getPath($< \texttt{UNIV}, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} >, < \texttt{UNIV}, \mathsf{a}, \mathsf{b}, \mathsf{e} >, < \texttt{CONT}, \texttt{CONT}, \mathsf{e} >$). The two constants of type link represent the top-level object and the illegal reference. `CONT, UNIV, SELF` denote symbolic references to the container object, the top-level object and the current object respectively. The functions container and behaviour return a link to the container object and the behaviour script of the object respectively. part is a function to obtain the link to an object which is part of the current object. getLink returns a link to an object (the top-level object, container object, the object itself or one of the contained objects) depending on the contents of the argument.

An object can respond to seven actions - new, create, destroy, exist, set, get, eval. All of them take as their first argument a path. When the object is created (Axiom (1)) a link to the container object, the *self* path of the container object and the name of the object itself are passed as arguments. The behaviour script is set to evaluate to an empty list, and the set of parts is initialised with the empty set.

Object object:LINK
Sorts

LINK, NAME, SCRIPT

with

interpret(SCRIPT, SEQ_{NAME}) : SEQ_{NAME}
toBehaviour(SEQ_{NAME}) : SCRIPT
newObj(LINK, NAME)
deleteObj(LINK, NAME)
getPath(SEQ_{NAME}, SEQ_{NAME}, SEQ_{NAME})
shift(SEQ_{NAME}, SEQ_{NAME}, SEQ_{NAME})

Constants

`Universe`, `NIL` : LINK
`CONT`, `UNIV`, `SELF` : NAME

Functions

container : LINK
behaviour : SCRIPT
self : SEQ_{NAME}
part(NAME) : LINK
getLink(NAME) : LINK

Actions

BEG(LINK, SEQ_{NAME}, NAME)
new(SEQ_{NAME}, SEQ_{NAME})
create(SEQ_{NAME})
destroy(SEQ_{NAME})
exist(SEQ_{NAME}, SEQ_{NAME})
set(SEQ_{NAME}, SCRIPT)
get(SEQ_{NAME}, SCRIPT)
eval(SEQ_{NAME}, SEQ_{NAME})

Figure 2.8: Axiomatisation of the Object Model (Part I)

Axioms

$$\mathsf{BEG}(x,s,n) \longrightarrow \mathsf{container} = x \wedge \mathsf{self} = s < n > \wedge$$

$$\mathsf{behaviour} = \mathsf{toBehaviour}(<>) \wedge \mathsf{part}(y) = \texttt{NIL} \tag{2.1}$$

$$\mathsf{self} = s \wedge \mathsf{getPath}(p,s,<>) \longrightarrow$$

$$(\mathsf{new}(p, s < n >) \wedge getLink(n) = \texttt{NIL} \longrightarrow \mathsf{newObj}(l,n)) \wedge \tag{2.2}$$

$$(\mathsf{create}(p)) \wedge \tag{2.3}$$

$$(\mathsf{destroy}(p) \longrightarrow (\mathsf{part}(y) = t \wedge t \neq \texttt{NIL} \longrightarrow t.\mathsf{destroy}(<>))) \wedge \tag{2.4}$$

$$(\mathsf{exist}(p,s)) \wedge \tag{2.5}$$

$$(\mathsf{set}(p,x) \longrightarrow \mathsf{X}(\mathsf{behaviour} = x)) \wedge \tag{2.6}$$

$$(\mathsf{behaviour} = x \longrightarrow \mathsf{get}(p,x)) \wedge \tag{2.7}$$

$$(\mathsf{behaviour} = x \wedge \mathsf{interpret}(x) = y \longrightarrow \mathsf{eval}(p,y)) \tag{2.8}$$

$$\mathsf{self} = s \wedge \mathsf{getPath}(p,s,<n>q) \wedge \mathsf{getLink}(n) = l \wedge l \neq \texttt{NIL} \longrightarrow$$

$$(\mathsf{new}(p,x) \longrightarrow l.\mathsf{new}(q,x)) \wedge \tag{2.9}$$

$$(\mathsf{create}(p) \longrightarrow l.\mathsf{create}(q)) \wedge \tag{2.10}$$

$$(\mathsf{destroy}(p) \wedge q \neq <> \longrightarrow l.\mathsf{destroy}(q)) \wedge \tag{2.11}$$

$$(\mathsf{destroy}(p) \wedge q = <> \longrightarrow \mathsf{deleteObj}(l,n) \wedge l.\mathsf{destroy}(q)) \wedge \tag{2.12}$$

$$(\mathsf{exist}(p,x) \longrightarrow l.\mathsf{exist}(q,x)) \wedge \tag{2.13}$$

$$(\mathsf{set}(p,x) \longrightarrow l.\mathsf{set}(q,x)) \wedge \tag{2.14}$$

$$(\mathsf{get}(p,x) \longrightarrow l.\mathsf{get}(q,x)) \wedge \tag{2.15}$$

$$(\mathsf{eval}(p,x) \longrightarrow l.\mathsf{eval}(p,x)) \tag{2.16}$$

$$\mathsf{self} = s \wedge \mathsf{getPath}(p,s,<n>q) \wedge \mathsf{getLink}(n) = \texttt{NIL} \wedge \neg\mathsf{object}(l) \longrightarrow$$

$$(\mathsf{new}(p,x) \longrightarrow \mathsf{newObj}(l,n) \wedge l.\mathsf{new}(q,x)) \wedge \tag{2.17}$$

$$(\mathsf{create}(p) \longrightarrow \mathsf{newObj}(l,n) \wedge l.\mathsf{create}(q)) \wedge \tag{2.18}$$

$$(\mathsf{destroy}(p) \wedge q \neq <> \longrightarrow \mathsf{newObj}(l,n) \wedge l.\mathsf{destroy}(q)) \wedge \tag{2.19}$$

$$(\mathsf{destroy}(p) \wedge q = <>) \wedge \tag{2.20}$$

$$(\mathsf{exist}(p,\texttt{NIL})) \wedge \tag{2.21}$$

$$(\mathsf{set}(p,x) \longrightarrow \mathsf{newObj}(l,n) \wedge l.\mathsf{set}(q,x)) \wedge \tag{2.22}$$

$$(\mathsf{get}(p,x) \longrightarrow \mathsf{newObj}(l,n) \wedge l.\mathsf{get}(q,x)) \wedge \tag{2.23}$$

$$(\mathsf{eval}(p,x) \longrightarrow \mathsf{newObj}(l,n) \wedge l.\mathsf{eval}(q,x)) \tag{2.24}$$

$$\mathsf{self} = s \wedge \mathsf{newObj}(l,n) \longrightarrow l.\mathsf{BEG}(object,s,n) \wedge \mathsf{X}(\mathsf{part}(n) = l \wedge \mathsf{object}(l)) \tag{2.25}$$

$$\mathsf{deleteObj}(l,n) \longrightarrow \mathsf{X}(\mathsf{part}(n) = \texttt{NIL} \wedge \neg\mathsf{object}(l)) \tag{2.26}$$

$$\mathsf{getLink}(\texttt{CONT}) = \mathsf{container} \tag{2.27}$$

$$\mathsf{getLink}(\texttt{UNIV}) = \texttt{Universe} \tag{2.28}$$

$$\mathsf{getLink}(\texttt{SELF}) = \mathsf{object} \tag{2.29}$$

$$n \neq \texttt{CONT} \wedge n \neq \texttt{UNIV} \wedge n \neq \texttt{SELF} \longrightarrow \mathsf{getLink}(n) = \mathsf{part}(n) \tag{2.30}$$

$$n \neq \texttt{UNIV} \longrightarrow \mathsf{getPath}(<\texttt{UNIV}>p, <n>q, <n>q) \tag{2.31}$$

$$n \neq m \wedge \mathsf{shift}(<n>p, <m>q, r) \longrightarrow \mathsf{getPath}(<n>p, <m>q, r) \tag{2.32}$$

$$\mathsf{getPath}(p,q,r) \longrightarrow \mathsf{getPath}(<n>p, <n>q, r) \tag{2.33}$$

$$\mathsf{shift}(p, <\texttt{CONT}>q, r) \longrightarrow \mathsf{shift}(<n>p, q, r) \tag{2.34}$$

$$\mathsf{shift}(<>, q, q) \tag{2.35}$$

Figure 2.9: Axiomatisation of Object Model (Part II)

An objects responds to an action itself if the path obtained from the getPath predicate is an empty list (Axioms (2)-(8)). Otherwise it forwards the request to the object specified in the first element of the path (Axioms (9)-(24)). The new action creates a sub-object with a fresh name. The path to the new object is returned in the second argument. Most of the activities in create, destroy, exist take place in the container object as they are management actions which have to be carried out by the manager (e.g. the container object). Hence the axioms (3)-(5) are trivial, apart from the fact that the destruction of an object also causes the destruction of all sub-objects (axiom (4)). Only the exist action actually does something. It binds it's second argument to the *self* path as an indication that the object exist. We use *self* and `NIL` instead of two additional constants `True` and `False`. This is safe because the object associated with the *self* path always exist while there is never an object with the link `NIL`. The other three actions deal with the object state and therefore all activities take place in the object itself (Axioms (6)-(8)). The set action sets the behaviour script of the object. The script can be retrieved with get or evaluated with eval.

If the object, which the message is to be forwarded to, doesn't exist, it is created (Axioms (17)-(24)). The creation of an object is initiated by invoking the BEG action. The set of parts and the global predicate object are updated. delete reverses that action. The method of implicitly creating objects when they are accessed has some advantages. It enables us to specify the model without error or exception handling. An error or exception occurs when attempts are made to:

- create an object that already exists
- access an object that doesn't exist
- destroy an object that doesn't exist

In our model the attempt to create an object that already exist is ignored. Similarly a request to an object creates that object in case it doesn't exist. An exception from this rule is the exist action which of course doesn't create any objects. Note that in *all* other cases all objects on the path to the object that is to be accessed are created if they don't exist in order to maintain the tree structure of the decomposition hierarchy.

2.2.2 Messages and Recursion

Our version of the object model inevitably affects the way messages are exchanged. The commonly used statement to send a message to an object

$$result = object.message(arg_1, arg_2, \ldots, arg_n)$$

no longer exists in that form. For every message that an object can receive, a sub-object exists. We shall call these sub-object *method-objects* or *methods* as their meaning is similar to methods as we know them from languages like *SmallTalk*. Figure 2.10 shows how methods are translated into method objects. Instead of sending a message to the object in this model we invoke the interpretation of the behaviour script of the appropriate method - the method-object whose name is identical with the message identifier. The set of methods (e.g. the set of messages an object can receive) does not have to be constant. Method-objects may be destroyed thereby depriving the object of receiving particular messages or new methods may be created - extending the set of receivable messages. It is also possible to change the behaviour script of a method-object.

Message Arguments

The arguments of a message are handled in a special way in our model.. Taking into consideration that the arguments of a message should be assigned to local variables of the method, we can treat them as such. Local variables become parts of the method-object. The transfer of the arguments then takes place before the actual method is called by creating the sub-objects for the formal parameters and assigning to them the values of the arguments. The original statement for sending

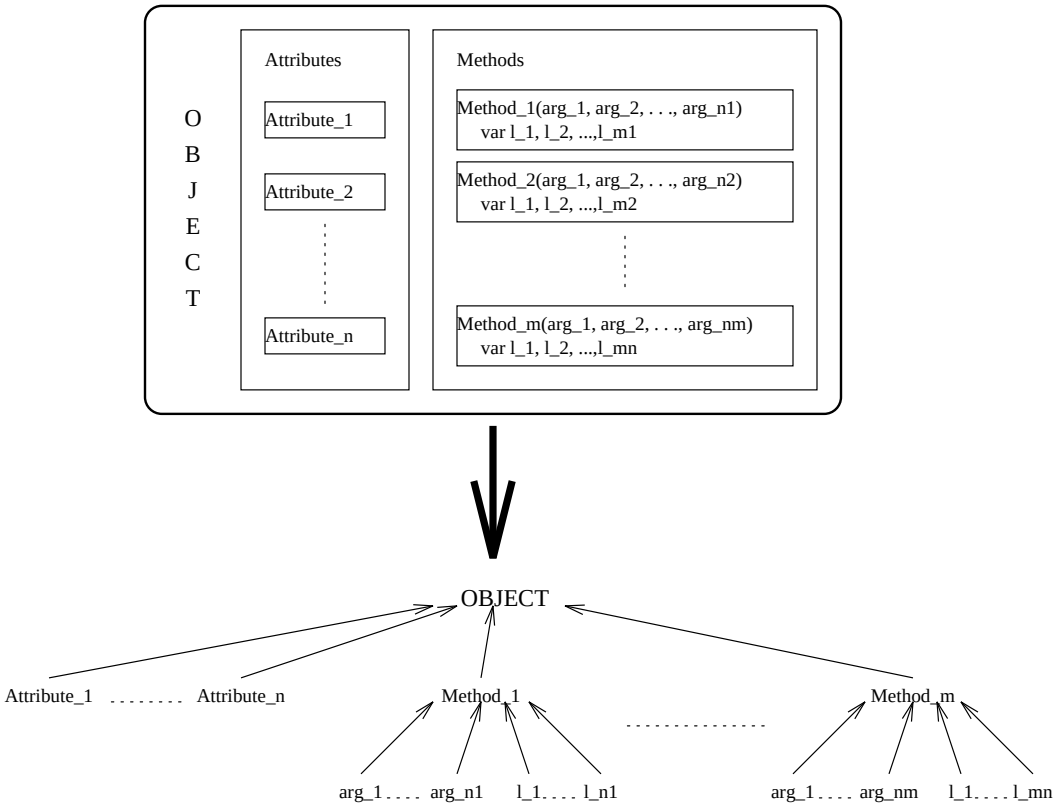

Figure 2.10: Methods as Part of the Decomposition Hierarchy

a message can be translated into the following sequence of statements (assuming that the formal parameters of the method are named $par_1 \ldots par_n$):

$object/message/par_1 = arg_1$
$object/message/par_2 = arg_2$
...
$object/message/par_n = arg_n$
result=$object/message$

The first set of statements transfers the arguments to the parameter-objects of the method. The last statement then invokes the method.

The approach taken has some side effects which are not immediately obvious. What happens if the method is invoked recursively? In our model the same script can be interpreted more than once at the same time. So deadlock doesn't occur if we have a recursion. The method object can accept requests for evaluation (e.g. interpreting the behaviour script) at any time. The problem with recursion lies somewhere else - in our translation of local variables. The local variables of a method (e.g. the arguments of a message) have become sub-objects of the method-object. So when the arguments of the recursive invocation are assigned to these objects the previously assigned arguments (of the earlier invocation) are overwritten! To prevent this an object creates a copy of itself and all its sub-objects when it receives a request for evaluation. The newly created object then evaluates the behaviour script. Accesses to sub-objects are received by the copies of those. Objects outside the tree are accessed as usual. This approach is a novelty in the design of object models. It is essentially the ordinary procedure invocation mechanism applied to our model where the procedure invocation corresponds to the evaluation of a method object. The one layer of decomposition in procedures comprises the arguments of the invocation and the static and dynamic variables. Unlike procedures our method objects have more than one layer of decomposition. The

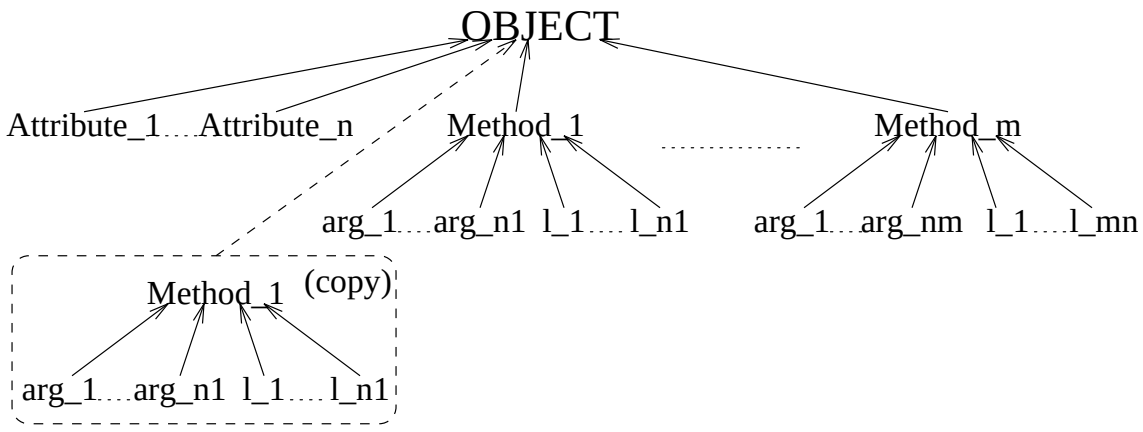

Figure 2.11: Method Invocation

example in Figure 2.11 demonstrates the invocation of a method. It can be seen how the additional subtree is created and which objects are accessed. The dotted arrow indicates that this part of the hierarchy is invisible to the other objects further up in the hierarchy.

We could have a less eager strategy when creating the copy of the method object. We might delay the creation of the copies of the sub-objects until they are being accessed by the copy of the method object. However, in order to prevent the modification of arguments of the method invocation we would also have to create the copies when sub-objects of the original method object are modified.

The Transparency of the Subtree

The subtree that is created when evaluating a method object is part of the decomposition hierarchy. For all the objects in it the whole process of creating copies is transparent. On the other hand objects outside this subtree can't access the objects in it. The subtree is invisible to them and their requests go to the original objects, making it possible to start further evaluations at the same time. When the interpretation of a script finishes the subtree is removed again. During this process the removal of any sub-subtrees is delayed if there are still scripts being interpreted in objects that reside in those parts of the tree. Thereby we prevent the destruction of the environment in which those script interpretations are taking place. Once the sub-tree is removed all changes to the objects in it are lost. Recall, however, that changes to objects further up in the hierarchy remain. This is important as otherwise the attributes of an object couldn't be changed permanently. The attributes are parts of the object. They reside on the same level of decomposition as the method-objects. Hence when a method-object wishes to access an attribute of the object it is part of this request is handled by the original attribute-object.

2.2.3 Synchronisation and other Concurrency Issues

In any concurrent system synchronisation plays an important role. Without means of synchronising activities any such system would behave chaotic. Methods for synchronisation are not specified in the presented object model. We could attempt to implement synchronisation methods on top of the presented model. But it turns out that without any low-level synchronisation this is impossible. As in the case of recursion the problems are again caused by the method modelling. When we transfer the arguments to a method (as it was described above) we surely wish that no other object interferes with that process. So we need the concept of a transaction. But we can't model transactions with the means we have. If we for instance tried to coordinate the access to the method object by granting a permission we would only shift the problem to the object which grants the permission. To resolve this conflict we have to introduce two more actions into our

specification - *lock* and *unlock*. Any method invocation (a sequence of statements to transfer the arguments and the actual call) must enclosed between a *lock* and *unlock* action and

lock $object/message$
$object/message/par_1 = arg_1$
$object/message/par_2 = arg_2$
...
$object/message/par_n = arg_n$
$result = object/message$
unlock $object/message$

Any object is either in a locked or unlocked state. If an object is locked it delays the execution of all *lock* requests. If it is unlocked a *lock* request is answered and the object goes into the locked state.

We have to be careful with recursion again. When a method object is invoked in its locked state (as in the example) recursion would yield to deadlock. However, from the preceeding section we know that the invocation is actually sent to a copy of the method object and hence all the messages that are sent to the object from lower levels of the hierarchy (i.e. from inside the method) are being processed by the copy. This copy thus must be set into an unlocked state after its creation. This approach by default prevents other objects from invoking the method, as the original method object is still in its locked state. This can easily be overcome though. The method itself just has to explicitly unlock the original method object. Of course the *unlock* statement in the caller script (see above) is now redundant and must be ignored. This approach to synchronisation is quite elegant as the control over the degree of concurrency is determined by the method objects themselves.

The reliability of the information we retrieve from objects or from the system is another concurrency issue. We have multiple objects potentially accessing the same object. Any information gathered from such an communication can be invalid already by the time it is received: An existent object might have been deleted in the meantime, the behaviour script might have changed etc. In general we can make no assumption about the validity of such information. But in many cases this is needed. However, with the transaction concept we have a powerful tool to solve these problems. If we always first use a *lock* before we access an object we can make sure that the information we acquire is valid at least until we unlock that object again.

2.2.4 Inheritance

The object model in its current state doesn't incorporate inheritance. There is no doubt that the notion of inheritance is essential for any object-oriented design and one should be able to express it in the language. Furthermore the *inheritance anomaly* is a major problem faced by many concurrent object-oriented systems. The notion of inheritance clashes with the methods of synchronisation. One of the main concerns when devising the model was to overcome this anomaly. So why have we not included inheritance in our model? There are two main reasons for this:

- Inheritance is a very complex issue. There exist a variety of approaches. Building it into the model ultimately restricts us to one particular approach.
- Inheritance can be modelled on top of the presented model.

Both reasons may not be entirely convincing. Inheritance is a key feature of object-oriented systems. It should be built in - if for nothing else but efficiency reasons. Here we will only present some of the underlying ideas for modelling inheritance in a system like ours.

The Inheritance Anomaly

The inheritance anomaly [Fro92, JA92, KL93, BY87, YM93] arises when objects in an concurrent object-oriented system have an explicitly specified synchronisation policy. It's called an *anomaly*

because in general there are no problems in applying the inheritance mechanisms of an sequential system in a concurrent system while the inheritance of synchronisation policies gives rise to very difficult problems. There has been a lot of research in this area but so far no satisfactory solution has been reached. Here we shall only provide a brief account of the nature of the problem. For a detailed analysis of the problem (that occurs in different flavours with varying complexity) we refer to the excellent paper by Yonezawa [YM93].

A synchronisation policy states which methods of an object can be invoked at a given time. It is usually specified using conditions on the state variables of the object for enabling or disabling particular methods. If an object receives a message that invokes a method that is not enabled this message is rejected. Alternatively the sender is blocked and the message is queued until further messages arrive that change the object's state such that the requested method invocation becomes enabled. There are two ways of specifying the synchronisation policy. The first is to attach it to the methods, i.e. each method defines for itself under which conditions it can be invoked. In this case the synchronisation policy is inherited with the method. The second way is to have a special synchronisation code attached to each object, specifying the synchronisation policy for all methods in the object. This piece of code is inherited separately.

Unfortunately we observe that the inheritance of synchronisation policies is pretty useless. In both of the above mentioned approaches even the slightest additions and modifications that are made in the child-objects with respect to the parent object force us to almost completely re-define the synchronisation policies. In general it can be said that the problem arises because the synchronisation policies by their very nature involve the specification of inter-dependencies between methods, between object states and between both. These complex inter-dependencies are disturbed easily if a new element (such as an additional method or state variable) enters the scene or if existing elements are being modified (e.g. re-definition of a method).

The described way of specifying synchronisation policies presupposes an object model in which the objects have a constant set of state variables and methods. We can't specify the inter-dependencies if we don't know about the elements. From this it should be clear that these approaches can't be applied in our model where methods and state variables have been unified and can moreover be added, removed and modified dynamically. In fact it also follows that the inheritance anomaly doesn't arise in our model. We don't have any explicit code for specifying synchronisation policies. The best way to overcome the inheritance anomaly is not to find more and more sophisticated ways of writing the synchronisation code but not to having to write such code in the first place. This view was also put forward in a recent paper by Meseguer [Mes93]. Of course it's not *that* easy to eliminate the problem for we clearly *have* to write code that synchronises the activities in our system. How this is done in a system like ours with a dynamic notion of inheritance is still a topic of research.

Delegation

Inheritance in our model is best described as delegation. It is the most suitable approach in the kind of system we look at. Other authors ([BY87, KL89]) have come to the same conclusion when investigating inheritance in highly dynamic object-oriented systems. We don't have the explicit notion of classes. Inheritance can occur between any objects. Furthermore in the notion of delegation instantiation is a special case of inheritance. It turns out that some of our model's features make it particularly suitable for incorporating delegation without great difficulty. How does delegation work?

> If an object receives a message it doesn't understand it forwards this message to it's parent object. If this doesn't understand it either it forwards the message to its parent. Eventually the message arrives either at the top object of the abstraction hierarchy and is rejected or one of the parents in the chain accepts it. In this case it executes the associated method. But the execution takes place in the context of the object the message was send to originally. In case of a multiple inheritance the message is only rejected if all alternatives fail.

We recall that in our model the methods are sub-objects of the receiver object. Thus if a method corresponding to a message doesn't exist the sub-object with the name of the method doesn't exist. Hence "not understanding a message" means that a container object can't forward the message to one of its parts because this part doesn't exist. In the current version of our model this part is then created. When we incorporate delegation we forward the message to the parent object instead (in case of multiple inheritance there can be several). We must distinguish however between normal messages and delegated messages. If a delegated message is accepted the action has to take place in the context of the original object. If it is finally rejected we proceed as in the current model (i.e. creating the object that was to be accessed). When we follow this idea further we eventually end up with a notion of inheritance that is slightly different from delegation and was introduced by A. Yonezawa in [BY87] - the *Recipe-Query Scheme*. In order to execute a method in the context of another object (that is precisely what has to happen when a parent object reacts to the message on behalf of the original receiver) the body of the method (the *recipe*) is transferred to the original object. So actually the original message is not forwarded to the parent object but replaced by a query for the body of the method (i.e. the behaviour script).

2.2.5 Destruction of Objects

The proposed object model results in a very high rate of object creation. Without some means of destructing objects any system would slow down considerably and eventually crash. From the user's point of view a *logical* destruction of objects is sufficient, i.e. all visible references to the object are removed. All the features of the model that require the destruction of objects could be implemented this way. However, for the above mentioned reasons we additionally require the actual physical destruction of the objects, i.e. the memory and processes allocated to them have to be free'ed.

Garbage collection

A common solution to this problem is to use *garbage collection* [BC92]. From the model's point of view only the logical destruction has to be specified while the actual physical destruction remains transparent - which is an obvious model theoretic advantage. In the implementation of the system we add a garbage collector component. This component keeps track of all the the logically (and hence physically) created and logically destructed objects. It physically destructs objects at a convenient time determined by some algorithm. Garbage collection is a complex problems, even more so in the context of a distributed system where the garbage collector itself has to be distributed.

Unfortunately, for various reasons, it is not sensible to use garbage collection as a means for destroying objects in our model. The logical destruction of objects is a rather complex process (as we shall see later). It requires insight into the object model and it affects the behaviour of the system in general. By contrast the actual physical destruction is trivial. Having a garbage collection algorithm only makes sense if the object model can be kept free from any complicated mechanisms concerning object destruction. Hence, while we *could* have a garbage collection component in our system it wouldn't give us any gain from the model theoretic point of view.

The envisaged implementations of systems based on the model were to be carried out in environments that make it difficult to integrate garbage collection. On the other hand, explicit and direct physical destruction of objects is a straightforward task. This too makes the use of garbage collection a less favourable approach.

The Logical Destruction of Objects

The logical destruction of objects in our model is an issue that needs some close examination. What exactly should happen if the destruction of an object is requested?

Let us assume for a moment that our system was in a *quiet* state, i.e. no object is being evaluated and no messages are being passed. If an object receives the *destroy* request it is immediately

decoupled from the container object, i.e. it disappears from the view of all the objects in the decomposition hierarchy that are situated in different branches or at higher levels of the same branch. At this point we can already acknowledge the destruction of the object to whoever sent the request. In order to maintain the connectivity axioms of the hierarchy (cf. Figure 2.1) the object has to destroy all its sub-objects by sending them *destroy* requests. It can do this sequentially or in parallel, i.e. sending all the requests and then waiting for all the confirmations. After this the object can physically destruct itself. Destruction of objects is thus a recursive process over the structure of the decomposition hierarchy.

The reader may have noticed that we make some assumptions about who is requesting the destruction of an object. The destruction, as any other operation, is acknowledged to the sender of the request. Thus, in order to avoid deadlock, that object must still exist at the time of acknowledgement. Hence objects are not allowed to logically destruct themselves or any objects at higher levels of the same branch of the decomposition hierarchy. They're also not allowed to perform these actions indirectly, i.e. telling another object to do it on their behalf, unless they don't require confirmation of the successful completion of the activity. It should be obvious that at least the latter constraint can't be enforced by the system and it is thus up to the programmer to make sure that his/her programs comply with these rules. The rules themselves are sensible. It's a common policy in object-oriented design that objects should be destroyed by the same object that created them. Plainly an object can't create itself nor can a sub-object create its container object. Hence this design policy forces the objects to comply with the rules of our system.

Destruction of objects in a system like ours, where objects are being evaluated and messages are being sent all the time, is a more complex issue than in a quiet system. First there are invisible objects - whenever an object is evaluated a copy of that object and its sub-objects is created as a part of the hierarchy that is invisible to the rest of the system. If an object is to be destroyed we must ensure that those sub-objects are destroyed together with the proper sub-objects. The sub-objects which have been created for evaluation are automatically destroyed once the evaluation has finished. We thus only keep in the container object a counter of the number of these objects and decrement it when they acknowledge their destruction. We can apply the same method to the proper sub-objects, i.e. when a proper sub-object is created we increment the counter and when it acknowledges its destruction we decrement it. The destruction of the object itself is delayed until the counter reaches zero and only then we send back the confirmation.

The destruction of an object depends on the termination of the evaluation of all invisible sub-objects. We therefore have to make sure that the process of destruction doesn't interfere with the evaluation, i.e. to the evaluating object the view of the decomposition hierarchy must remain unaffected. That's impossible to ensure as other objects already may have been destructed. We can easily guarantee that objects waiting for destruction confirmations are still able to process further incoming messages, but once they are destroyed this is obviously impossible. A solution to the problem is to wait until all the objects in the sub-tree are quiet and not to destroy any objects before that. However, new problems arise in such an approach, like messages that have been sent but haven't reached there destination yet. Maybe further research in this area can provide an elegant solution. The encountered problems are by no means system-specific. It's not a flaw in our model but rather we deal with a class of general problems in concurrent and distributed systems. The proposed approach using counters solves at least some of them.

Chapter 3

Design of the Language

Object models have always been closely related to particular languages. When designing a language the merits and flaws of the underlying model can become apparent. Ideally the language is not an issue itself - if it has been designed carefully the relation to the model should be immediately recognisable. Only then programming in the language means thinking in terms of the underlying model rather than thinking in terms of the language itself. The relation between model and language is established by the semantic description of the language. Without a precise semantics a language is defined by the various compilers that have been devised for it. In this chapter we formally and informally specify both syntax and semantics of a language that is based on our model.

3.1 Language Syntax and Informal Semantics

In this section we define a language based on the developed object model. The syntax definition of the language is followed by an example and an informal description of the semantics of the language constructs.

3.1.1 Syntax Definition

To define the syntax of the language we use a variant of BNF. Terminal symbols are emphasised; non-terminal symbols have upper case initials. The meta-syntactic constructs used are:

```
|          alternative
{expr}     zero or more
```

A program is defined as *Block*. A *Block* consists of a sequence of *Statements*. Blocks can also be created from references or retrieved from objects. Blocks in the object model are called *behaviour scripts*.

$$
\begin{array}{lcl}
Block & = & [\ \{Statement;\}\ ] \\
 & | & @Obj \\
 & | & \#Obj
\end{array}
$$

There are four statements defined in the language - the creation and destruction of an object, the assignment of a block to an object and a statement to return a reference from a block that is being executed.

$$
\begin{array}{lcl}
Statement & = & +Obj \\
 & | & -Obj \\
 & | & Obj{=}Block \\
 & | & !Obj
\end{array}
$$

References to objects are represented as paths. These paths indicate the objects position in the decomposition hierarchy. References can be constructed using operations on paths (including concatenation). References can also be compared or be the tag for alternatives. The result of creating an object with a fresh name, testing the existence of an object or evaluating a block associated with an object, is a reference too.

$$
\begin{array}{lcl}
Obj & = & SimpleRef \\
 & | & Obj/Obj \\
 & | & \textbf{head}\ Obj \\
 & | & \textbf{tail}\ Obj \\
 & | & \textbf{front}\ Obj \\
 & | & \textbf{last}\ Obj \\
 & | & Obj{==}Obj \\
 & | & Obj\textbf{?}Obj\textbf{:}Obj \\
 & | & \textbf{new}\ Obj \\
 & | & \textbf{exist}\ Obj \\
 & | & \textbf{*}Obj \\
 & | & (Obj)
\end{array}
$$

The operators of the language concerning paths have precedences. The unary operators have the highest precedence. Of the binary operators / has the highest precedence, followed by == and ?:. / and == are both right-associative. Parenthesis can be used to override the default precedences.

Paths can be relative or absolute. Special identifiers are used for the container object (denoted by **..**), the object itself (**.**) and the top-level object (**_**). A global reference starts with the reference to the top-level object or with the global self-reference (**&**). The invalid (i.e. empty) reference is denoted by **NIL**.

$$
\begin{array}{lcl}
SimpleRef & = & Name \\
 & | & \textbf{..} \\
 & | & \textbf{.} \\
 & | & \textbf{\_} \\
 & | & \textbf{\&} \\
 & | & \textbf{NIL}
\end{array}
$$

3.1.2 Example

To demonstrate the usage of the language we use the well known example of a *Primes Sieve* (cf. for instance [MDK93, MDK94]). The decomposition hierarchy is sketched out in Figure B.2 and the program itself is shown Figure 3.1. Dotted arrows have been used to indicate some of the references to objects.

The numbers to be sieved are generated by a number generator. The actual sieve is a pipeline. Each stage tests whether the received number can be divided by the prime associated with the stage. The number is passed to the next stage if the division is not possible. The last stage of the pipeline has a different functionality. If a number gets to this stage then it is a prime. It is sent to the output and a new pipeline stage with that number is inserted. The number generator has to create the numbers in ascending order starting with the smallest prime.

The syntax and semantics of **lock** and **unlock** statements for realizing a transaction concept are not being described in this paper. Together with some changes in the evaluation procedure these modifications will be subject of a forthcoming paper. Both extensions are necessary to cope with recursion and synchronisation. For a discussion on these issues we refer to the preceeding chapters.

3.1.3 Informal Semantics

The only data structures appearing in the language are *object references* and *blocks*. Each object is part of the decomposition hierarchy. The global pathname of an object identifies its position

```
[
PrimesSieve=[
    NumberGen/generate=[
        lock *(../output);
        !NIL;                           //achieve maximum throughput
        *(../output)/value=@*(../value);//prepare output
        tmp=@**(../output);             //output the number
        unlock *(../output);
        ../value=@*(*(../value)/inc2);  //increment
        tmp=@*(../(last &));            //next iteration
        ];
    tmp=@(new Sieve);                   //create the first sieve
    NumberGen/output=@*tmp/sieve;       //bind the number generator output to it
    NumberGen/value=@_/Numbers/3;       //initialize the generator
    *tmp/sieve=[                        //script block for the sieve
        ../next=@(new ..);              //create a new sieve
        *(../next)/(last &)=#(../(last &));
        ../prime=@*value;               //alter the current sieve
        -../output;                     //      "
        ../(last &)=[                   //      "
            lock *(../next)/(last &);
            !NIL;                       //achieve maximum throughput
            *(../next)/(last &)/value=@*value;//in case of forwarding
            *value/mod/value=@*(../prime);    //prepare modulo test
            tmp=@(*(*value/mod)==_/Numbers/0) ? NIL : *(*(../next)/(last &));
                //forward the number if it can't be divided
            unlock *(../next)/(last &);
            ];
        lock *(../output);
        !NIL;
        *(../output)/value=@*value;     //prepare output
        tmp=@**(../output);             //output the prime
        unlock *(../output);
        ];
    *tmp/output=@_/Output;              //initialize the sieve
    tmp=@*NumberGen;                    //start the number generator
    !NIL;
    ];
tmp=@*PrimesSieve; //start the sieve
]
```

Figure 3.1: Example Program for Primes Sieve

in this hierarchy. Each object has an associated block. This block can be *set* and *retrieved*. An object can also be *evaluated*. During evaluation the object's block is executed in the context of the object. The block contains statements of the proposed language. The evaluation of a block results in an object reference. Thus when a block is evaluated exactly one *return* statement must be executed to return this reference. If no return of information is required a dummy reference has to be created and returned.

Object References

There are several ways to construct object references. One must distinguish between relative (local) and global references. A global reference starts with **_** which is the reference to the top level object, or **&** which denotes the global reference to the current object (the object in whose context the evaluation of the block is taking place). Relative references start with a **..** for the container object (the object the current object is part of), **.** for the object itself or a name. The parts of the objects (e.g. the sub-objects) can be accessed by their names. These references (or *paths* as we shall sometimes call them) can be concatenated using the / operator. The operator doesn't care about the nature of its two arguments. Hence it is possible to add a global reference to the end of a relative reference. Depending on the context such a reference occurs in, this can have various but well-defined side effects.

Further operations on paths are defined:

- **head** - the first element of the path, converted to a (single element) path
- **last** - the last element of the path, converted to a (single element) path
- **front** - all but the last element
- **tail** - all but the first element
- ref==ref - compares two paths
- ref?ref:ref - alternative

head, **last**, **front**, **tail** if applied to an empty path all return an empty path. Applied to a single element path **head** and **last** return this path while **front** and **tail** return an empty path. The fifth operation compares two references. If they are equal, that is all there elements are identical, the reference itself is returned. Otherwise the operation evaluates to **NIL**. The last operation in the list involves four references (the result is the fourth). If the first reference is not **NIL** the third reference is returned otherwise the fourth. In any case the other reference is *not* parsed. The reason for introducing this rather complex function is the need for modelling conditionals and branching in some way. This had to be done without using an additional data type (e.g. Boolean). So the result of such a operation had to be a reference. Actually we've laid the foundations for representing the type Boolean in our language. A simple test for equality returns references to the objects *true* and *false* depending on the result:

```
isEqual=Obj1==Obj2?_/T:_/F;
```

Operations on Objects

Applying the **new** operator to a path creates an object that is part of the object referred to by the path. It is ensured that the name of the object has not been used yet as a name for any other sub-object. The result is the complete path to the newly created object. The **exist** operation tests whether the object referred to in the reference exists. If so it returns the reference itself, otherwise **NIL**. The ***** operator is applied to a reference to evaluate an object and thus obtain a reference which can be used in place of any directly specified reference. A connection between the current object and the object referred to in that reference is established. The object is evaluated and the result (a path) is sent back.

There are only four statements in the block language. Two for creating/destroying objects (+ and −), one for returning a reference during the evaluation of a block (!) and one for assigning a block to an object. Blocks can be constructed with the [..] operator. They can also be created from paths. Applying the @ operator to a reference causes the reference to be converted to a block. If this block is evaluated in any context exactly this reference is returned as the result. A block can also be retrieved from an object using the # operator.

The statements of the language are executed sequentially. Intra-object concurrency arises because:

- several copies of an object can be created temporarily for evaluation (cf. Section 2.2.2). Although this is an instance of inter-object concurreny from the system's point of view, for the user it appears intra-object concurrency.
- Statements in the language frequently involve more than one access to objects. These accesses can potentially happen concurrently.

Intra-object concurrency from the user's point of view arises when the return statement of a block is not the last statement to be executed. In this case the sender object (or rather its block) can proceed with the evaluation without waiting for the termination of the evaluation of the receiver object. By making the return statement the first statement in a block maximum concurrency is achieved. This however only works if the result is a dummy reference or can be obtained easily in another way.

Accessing Objects

If an object is accessed that doesn't exist it is created. The associated block is initially set up to evaluate to an empty path. In a "proper" creation of an object this access is on the left hand side of an = statement. In these cases the associated block is immediately overwritten with the block obtained from the right hand side. Objects may also be created using two special statement from the language (+ and **new**). Accessing an object also means accessing all the objects on the path to it. Thus one access can create several objects. This is particularly important as the language doesn't optimise paths on its own.[1] If for instance a relative reference is followed by an absolute reference the relative part is not discarded. All the objects on the path are accessed, although this would not be necessary to access the object specified by the global reference. Hence additional objects might be created. An exception to this rule is the **exist** operator which doesn't create objects if they don't exist. Applied to the case of a relative reference followed by a global reference this means that the object referred to by the global reference is not found if any of the objects on the path of the relative reference don't exist.

3.2 Formal π-calculus Semantics

The reason for describing the semantics in terms of the π-calculus[MPW89, Mil91, Wal89] is that the calculus provides a convenient way of modelling aspects of concurrency and parallelism. These aspects are essential for the language. Having a well-founded calculus to describe exactly these gives us the opportunity to reason about the language in terms of the calculus. This process can then be supported by existing tools for the calculus. It is even possible to devise a first interpreter for the language using these tools. A program written in the language would be translated into the calculus and then executed.

The application of the π-calculus in describing the semantics of object-oriented languages was first introduced by D.Walker in [Wal91] and more recently in [Wal93]. In the field of concurrency and distributed systems S.Eisenbach and R.Patterson defined the semantics of the configuration language *Darwin* [EP93]. In further research this area by the author the semantics of more complex

[1] A recent change in the model overcomes this. Objects now always communicate via the shortest path.

features of the Darwinlanguage was defined. Together with a semantic definition of communication system features this led to the implementation of a translator from Darwininto π-calculus [RE94b].

Our research incorporates both aspects - object-orientation and concurrency. The π-calculus semantics combines the two. A brief introduction to the calculus can be found in Appendix A.1. We use a version of the π-calculus with an additional operator - *mismatch*. The extension of the calculus with that operator is described in Appendix A.2.

3.2.1 Auxiliary Definitions

First we define some abstract data types in terms of the calculus. These definitions we shall use later when defining the semantics of the language.

Paths

Pathnames are usually represented as lists. We need the five standard operations for lists:

$$
\begin{array}{rcl}
List(c) & \stackrel{\text{def}}{=} & c(u,v).u : [\texttt{EMPTY} \Rightarrow \overline{v}(\texttt{T}).List(c), \\
 & & \quad \texttt{ADD} \Rightarrow (\nu\ tl)(List(c,tl,v) \mid List(tl)), \\
 & & \quad \texttt{DESTROY} \Rightarrow \mathbf{0}, \\
 & & \quad \texttt{RELINK} \Rightarrow List(v)] \\
List(c,t,n) & \stackrel{\text{def}}{=} & c(u,v).u : [\texttt{EMPTY} \Rightarrow \overline{v}(\texttt{F}).List(c,t,n), \\
 & & \quad \texttt{ADD} \Rightarrow (\nu\ tl)(List(c,tl,v) \mid List(tl,t,n)), \\
 & & \quad \texttt{HEAD} \Rightarrow \overline{v}(n).\overline{t}(\texttt{RELINK},c).\mathbf{0}, \\
 & & \quad \texttt{DESTROY} \Rightarrow \overline{t}(\texttt{DESTROY},\texttt{NIL}).\mathbf{0}, \\
 & & \quad \texttt{RELINK} \Rightarrow List(v,t,n)]
\end{array}
$$

Figure 3.2: Definition of Lists in the π-caclulus

- create – create an empty list
- empty – test whether the list is empty
- add – add an item at the front of the list
- head – remove the head of the list and return it
- destroy – destroy the list

The *head* operation is destructive, that is the first element is actually removed from the original list. The definition of lists is shown in Figure 3.2.

We need to define some additional operations on lists (Figure 3.3). The first one is *Reverse*. As the name suggests it reorders the list in a way such that the first element becomes last, the second element becomes second last etc. and the last element becomes first. The first argument for *Reverse* is a link to the list. When the operation is finished the result is linked to the second argument. The operation is destructive, that is the original is destroyed. A derived form of *Reverse*, taking three arguments is used for appending the reverse of a list to the end of another list. It is invoked by the two-argument *Reverse* which supplies an empty list as the second list.

Secondly we need some means of comparing two lists. Two lists are equal if all there elements are equal. *Equal* takes two lists and does compare them. As a result the third list contains the prefix common to both lists and the original lists contain the remaining tails.

Most of the functions provided for lists so far are destructive. Hence it is essential to have a copy-operator to create an identical copy of a list. *Copy* creates a copy of the list whose link is provided as the first argument. The result is linked to the channel provided as the second argument. The previously defined *Reverse* is used in these equations.

$$
\begin{array}{rcl}
Reverse(l,c) & \stackrel{\text{def}}{=} & (\nu\ r)(List(r)\ \mid\ Reverse(l,r,c)) \\
Reverse(l,r,c) & \stackrel{\text{def}}{=} & (\nu\ q)(\overline{l}(\texttt{EMPTY},q).q(v). \\
 & & v:[\texttt{T}\Rightarrow \overline{r}(\texttt{RELINK},c).\mathbf{0}\ \mid\ \overline{l}(\texttt{DESTROY},\texttt{NIL}).\mathbf{0}, \\
 & & \quad \texttt{F}\Rightarrow \overline{l}(\texttt{HEAD},q).q(n).\overline{r}(\texttt{ADD},n).Reverse(l,r,c)]) \\
 & & \\
Equal(s,t,c) & \stackrel{\text{def}}{=} & (\nu\ r)(List(r)\ \mid\ Equal(s,t,r,c)) \\
Equal(s,t,r,c) & \stackrel{\text{def}}{=} & (\nu\ a,b)(\overline{s}(\texttt{EMPTY},a).\mathbf{0}\ \mid\ \overline{l}(\texttt{EMPTY},b).\mathbf{0}\ \mid \\
 & & a(u).b(v).u:[\texttt{T}\Rightarrow Reverse(r,c), \\
 & & \qquad\qquad \texttt{F}\Rightarrow v:[\texttt{T}\Rightarrow Reverse(r,c),\texttt{F}\Rightarrow Equal'(s,t,r,c)]]) \\
Equal'(s,t,r,c) & \stackrel{\text{def}}{=} & (\nu\ a,b)(\overline{s}(\texttt{HEAD},a).\mathbf{0}\ \mid\ \overline{l}(\texttt{HEAD},b).\mathbf{0}\ \mid \\
 & & a(u).b(v).([u=v].\overline{r}(\texttt{ADD},u).Equal(s,t,r,c) \\
 & & \qquad +[u\neq v].(\overline{s}(\texttt{ADD},u).\overline{t}(\texttt{ADD},\texttt{v}).Reverse(r,c)))) \\
 & & \\
Copy(s,t) & \stackrel{\text{def}}{=} & (\nu\ r,l)(List(r)\ \mid\ List(l)\ \mid\ Copy(s,t,r,l)) \\
Copy(s,t,r,l) & \stackrel{\text{def}}{=} & (\nu\ q)(\overline{s}(\texttt{EMPTY},q).q(v). \\
 & & v:[\texttt{T}\Rightarrow Reverse(l,t)\ \mid\ \overline{s}(\texttt{DESTROY},\texttt{NIL}).Reverse(r,s), \\
 & & \quad \texttt{F}\Rightarrow \overline{s}(\texttt{HEAD},q).q(n).\overline{r}(\texttt{ADD},n).\overline{l}(\texttt{ADD},n).Copy(s,t,r,l)])
\end{array}
$$

Figure 3.3: Auxiliary Operations on Lists

Sets

The decomposition hierarchy is modelled using sets. Each object defines for itself a set which

$$
\begin{array}{rcl}
Set(c) & \stackrel{\text{def}}{=} & c(v,m,r).v:[\texttt{NEW}\Rightarrow(\nu\ tail,n)(Set(c,tail,n,l)\ \mid\ Set(tail)\ \mid\ \overline{r}(n).\mathbf{0}), \\
 & & \qquad \texttt{ADD}\Rightarrow(\nu\ tail)(Set(c,tail,m,r)\ \mid\ Set(tail)), \\
 & & \qquad \texttt{REMOVE}\Rightarrow Set(c), \\
 & & \qquad \texttt{EXIST}\Rightarrow \overline{r}(\texttt{NIL}).\mathbf{0}\ \mid\ Set(c), \\
 & & \qquad \texttt{COPY}\Rightarrow Set(c)\ \mid\ Set(r), \\
 & & \qquad \texttt{DESTROY}\Rightarrow \mathbf{0}, \\
 & & \qquad \texttt{RELINK}\Rightarrow Set(r)] \\
 & & \\
Set(c,t,n,e) & \stackrel{\text{def}}{=} & c(v,m,r).v:[\texttt{NEW}\Rightarrow\overline{t}(\texttt{NEW},\texttt{NIL},r).Set(c,t,n,e), \\
 & & \qquad \texttt{ADD}\Rightarrow m:[n\Rightarrow Set(c,t,n,e), \\
 & & \qquad\qquad else\Rightarrow\overline{t}(\texttt{ADD},m,r).Set(c,t,n,e)], \\
 & & \qquad \texttt{REMOVE}\Rightarrow m:[n\Rightarrow\overline{e}(\texttt{DESTROY}).\mathbf{0}\ \mid\ \overline{t}(\texttt{RELINK},\texttt{NIL},c).\mathbf{0}, \\
 & & \qquad\qquad else\Rightarrow\overline{t}(\texttt{REMOVE},m,\texttt{NIL}).Set(c,t,n,e)], \\
 & & \qquad \texttt{EXIST}\Rightarrow m:[n\Rightarrow\overline{r}(e).\mathbf{0}\ \mid\ Set(c,t,n,e), \\
 & & \qquad\qquad else\Rightarrow\overline{t}(\texttt{EXIST},m,r).Set(c,t,n,e)], \\
 & & \qquad \texttt{COPY}\Rightarrow(\nu\ tail,el,p,q)(List(q)\ \mid \\
 & & \qquad\qquad \overline{t}(\texttt{COPY},m,tail).Set(c,t,n,e)\ \mid \\
 & & \qquad\qquad \overline{e}(q,\texttt{COPY},p).\overline{p}(m,el).\mathbf{0}\ \mid\ Set(r,tail,n,el)), \\
 & & \qquad \texttt{DESTROY}\Rightarrow(\nu\ q)(List(q)\overline{e}(q,\texttt{DESTROY},\texttt{NIL}).\mathbf{0})\ \mid \\
 & & \qquad\qquad \overline{t}(\texttt{DESTROY},\texttt{NIL},\texttt{NIL}).\mathbf{0}, \\
 & & \qquad \texttt{RELINK}\Rightarrow Set(r,t,n,e)]
\end{array}
$$

Figure 3.4: Definition of Sets in the π-calculus

holds all sub-objects. The objects are identified by their name which thus must be distinct for all sub-objects. We need the following operations:

- new – create an element with a fresh name.
- add – create an element if it doesn't exist yet.
- remove – remove an element.

- exist – test whether an object is already in the set.
- copy – create a copy of the set and all its elements.
- destroy – destroy the set and all objects in the set.

This specialised definition of a the general abstract data type *set* is expressed in terms of the π-calculus in Figure 3.4.

3.2.2 Blocks

For the definition of blocks we introduce a higher-order function (Figure 3.5). Unlike the previous

$$Block(c,P) \quad \stackrel{\text{def}}{=} \quad !(c(u,v).u : [\texttt{RELINK} \Rightarrow Block(v,P), \\ \texttt{EVAL} \Rightarrow v(self, cnt, sf, reply).P])$$

Figure 3.5: Higher-order Equation for Blocks

equations, where only channels appeared as arguments on the left hand side, here we have a process as the second argument. Blocks can be constructed in various ways. Their overall behaviour, however, is the same. They receive a command along a channel. There are only two commands, one for relinking the process (renaming the command channel) and one for evaluating the block. A block is evaluated in a context. This context (in form of a set a channels) is passed as an argument. The same block can be evaluated concurrently in many contexts. That's why the replication operator of the π-calculus (!) appears in the equation.

It should be noted that it is not necessary to use a higher-order equation. Instead we could have used a set of ordinary equations - one for each way to construct a block. But the higher-order equation allows a nice unified approach. Davide Sangiorgi developed a higher-order π-calculus (HOπ) in [San93a]. He also describes a way to compile HOπ to the original π-calculus. David Walker uses this new calculus in [Wal93] for defining the semantics of his concurrent object-oriented language. In our semantics the equation for blocks is the only equation where a higher-order notation is considerably more convenient. We therefore didn't use HOπ throughout the paper but rather stuck with the original calculus. However, a translation to HOπ is planned for the future.

3.2.3 The Object Process

The definition of the objects process (Figure 3.6) most directly affects the semantics of the language. The degree of concurrency and the synchronisation methods are determined by these equations. They're also the most complex equations of these definitions and all of the previously introduced definitions are needed.

An object has a name n which uniquely identifies it amongst the parts of its container object. Objects are only created by their container object. Any access to the object must be handled by the container object as the object is bound to its container via a private channel l. Other activities involving object management (like creating and destroying objects) are also carried out by the container object. It is therefore justified to call the container object the *manager* of its sub-objects.

For the efficient evaluation of statements in the proposed language objects maintain a local copy of their global identity in form of a list representing the path to the object from the root of the decomposition hierarchy. This path $self$ can easily be obtained by appending the object name n to the path of the container object s. Each object has got an associated block b which can be changed, retrieved or evaluated and is initially set up to evaluate to an empty list. A link to the top-level object is implicitly declared (the global link `Universe`). Figure 3.7 shows the structure of the object process.

$$
\begin{array}{rcl}
Obj(n,c,s,l) & \stackrel{\text{def}}{=} & (\nu\ sub,self,b,l_1,l_2)(Reverse(s,l_1)\ \mid\ List(l_2)\ \mid \\
 & & \overline{l_2}(\texttt{ADD},n).Reverse(l_1,l_2,self)\ \mid \\
 & & Block(b,(\nu\ l)(List(l)\ \mid\ \overline{reply}(l).\mathbf{0}))\ \mid \\
 & & Set(sub)\ \mid\ Obj(self,sub,b,c,l)) \\
 & & \\
Obj(sf,sb,bl,ct,c) & \stackrel{\text{def}}{=} & c(pt,a,l).(\nu\ p,d)(Path(sf,pt,p)\ \mid\ \overline{p}(\texttt{EMPTY},d).d(v). \\
 & & v:[\texttt{F}\Rightarrow \overline{p}(\texttt{HEAD},d).d(n).(\nu\ q)(Link(sb,ct,c,n,q)\ \mid\ q(w). \\
 & & \qquad w:[\texttt{NIL}\Rightarrow (\nu\ o)(\overline{sb}(\texttt{ADD},n,o).Obj(sf,sb,bl,ct,c)\ \mid \\
 & & \qquad\qquad Obj(n,c,sf,o)\ \mid\ \overline{o}(p,a,l).\mathbf{0}), \\
 & & \qquad\quad else\Rightarrow \overline{w}(p,a,l).Obj(sf,sb,bl,ct,c)]), \\
 & & \quad \texttt{T}\Rightarrow a:[\texttt{NEW}\Rightarrow (\nu\ r,s,t)(\overline{sb}(\texttt{NEW},\texttt{NIL},r).Obj(sf,sb,bl,ct,c)\ \mid \\
 & & \qquad\qquad Copy(sf,s)\ \mid\ Reverse(s,t)\ \mid \\
 & & \qquad\qquad r(na).\overline{t}(\texttt{ADD},na).Reverse(t,l)), \\
 & & \qquad \texttt{CREATE}\Rightarrow Obj(sf,sb,bl,ct,c), \\
 & & \qquad \texttt{DESTROY}\Rightarrow \overline{sb}(\texttt{DESTROY},\texttt{NIL},\texttt{NIL}).\mathbf{0}\ \mid \\
 & & \qquad\qquad \overline{sf}(\texttt{DESTROY},\texttt{NIL}).\mathbf{0}, \\
 & & \qquad \texttt{EXIST}\Rightarrow Copy(sf,l)\ \mid\ Obj(sf,sb,bl,ct,c), \\
 & & \qquad \texttt{COPY}\Rightarrow l(nct,nc).(\nu\ nsf,nsb)(\\
 & & \qquad\qquad \overline{sb}(\texttt{COPY},nc,nsb).Obj(sf,sb,bl,ct,c)\ \mid \\
 & & \qquad\qquad Copy(sf,nsf)\ \mid\ Obj(nsf,nsb,bl,nct,nc)), \\
 & & \qquad \texttt{SET}\Rightarrow Obj(sf,sb,l,ct,c), \\
 & & \qquad \texttt{GET}\Rightarrow \overline{bl}(\texttt{RELINK},l).Obj(sf,sb,bl,ct,c), \\
 & & \qquad \texttt{EVAL}\Rightarrow \overline{bl}(\texttt{EVAL},d).\overline{d}(sf,ct,c,l).Obj(sf,sb,bl,ct,c)]]) \\
 & & \\
Path(p,q,r) & \stackrel{\text{def}}{=} & (\nu\ l)(\overline{q}(\texttt{EMPTY},l).l(v). \\
 & & v:[\texttt{T}\Rightarrow \overline{q}(\texttt{RELINK},r).\mathbf{0}, \\
 & & \quad \texttt{F}\Rightarrow \overline{q}(\texttt{HEAD},l).l(v).\overline{q}(\texttt{ADD},v). \\
 & & \qquad ([v\neq \texttt{UNIV}](\nu\ s)(Copy(p,s)\ \mid\ Path'(p,q,r)) \\
 & & \qquad +[v=\texttt{UNIV}]\overline{q}(\texttt{RELINK},r).\mathbf{0})]) \\
 & & \\
Path'(p,q,r) & \stackrel{\text{def}}{=} & (\nu\ a,b)(\overline{p}(\texttt{EMPTY},a).\mathbf{0}\ \mid\ \overline{q}(\texttt{EMPTY},b).\mathbf{0}\ \mid\ a(u).b(v). \\
 & & u:[\texttt{T}\Rightarrow \overline{q}(\texttt{RELINK},r).\mathbf{0}, \\
 & & \quad \texttt{F}\Rightarrow v:[\texttt{T}\Rightarrow Shift(p,q,r), \\
 & & \qquad\qquad \texttt{F}\Rightarrow Path''(p,q,r)]] \\
 & & \\
Path''(p,q,r) & \stackrel{\text{def}}{=} & (\nu\ a,b)(\overline{p}(\texttt{HEAD},a).\mathbf{0}\ \mid\ \overline{q}(\texttt{HEAD},b).\mathbf{0}\ \mid\ a(u).b(v). \\
 & & ([u=v]Path'(p,q,r)+[u\neq v]\overline{p}(\texttt{ADD},u).\overline{q}(\texttt{ADD},v).Shift(p,q,r))) \\
 & & \\
Shift(p,q,r) & \stackrel{\text{def}}{=} & (\nu\ l)\overline{p}(\texttt{EMPTY},l).l(v). \\
 & & v:[\texttt{T}\Rightarrow \overline{q}(\texttt{RELINK},r).\mathbf{0}, \\
 & & \quad \texttt{F}\Rightarrow \overline{p}(\texttt{HEAD},l).\mathbf{0}\ \mid\ \overline{q}(\texttt{ADD},\texttt{CONT}).l(v).Shift(p,q,r)]) \\
 & & \\
Link(sb,ct,c,n,q) & \stackrel{\text{def}}{=} & n:[\texttt{CONT}\Rightarrow \overline{q}(ct).\mathbf{0}, \\
 & & \quad \texttt{SELF}\Rightarrow \overline{q}(c).\mathbf{0}, \\
 & & \quad \texttt{UNIV}\Rightarrow \overline{q}(\texttt{UNIVERSE}).\mathbf{0}, \\
 & & \quad else\Rightarrow \overline{sb}(\texttt{EXIST},n,q).\mathbf{0}]
\end{array}
$$

Figure 3.6: The Object Process

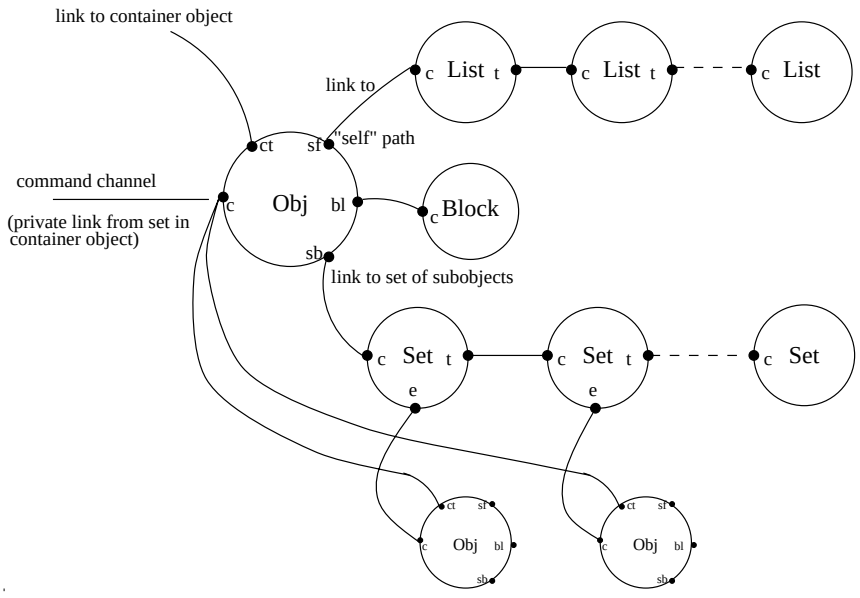

Figure 3.7: The Structure of the Object Process

Once created an object waits for commands passed along the private channel c (we are now referring to the second equation) from the container object. A command consists of three port names. The first one (p) is a link to an object reference. If this path is empty the object itself is accessed. Otherwise the object forwards the command to the object referred to in the path. This is done by removing the head of the path-list and according to its content (a special reference to the container object, the object itself, the top-level object or a name of a sub-object) accessing the appropriate object. The second port name in a command contains the operation to be performed. There are eight possible alternatives:

- new – create a new sub-object with a fresh name. As a result the third argument is bound to the path to the new object.
- create – create a new object.
- destroy – destroy the object.
- exist – test whether an object exist. As a result the third argument is bound to the path to the object or to an empty path if the object doesn't exist.
- copy – create an exact copy of the object and all its sub-objects. The third argument is the name of a channel for passing additional arguments.
- set – change the object's associated block. The third argument is a link to the new block.
- get – retrieve the object's associated block. The link to the block is passed along the channel specified in the third argument.
- eval – evaluate the object's associated block in the context of the object. The link to the resulting path is returned along the channel specified in the third argument.

If the object itself is accessed (e.g. the path-list is empty) these operations are performed.

The high degree of fine grain concurrency in the equations (achieved by using parallel instead of sequential composition) is in some cases even necessary. This can be seen when we look at the case where the command is forwarded to the object itself. To make this work the object has to be ready to receive a command along its command channel. The parallel composition in this case

prevents the process from deadlocking. Parallel composition is also needed to cope with indirect recursion.

3.2.4 Language Constructs

We can now finally define the semantics of all the constructs in the language in terms of the π-calculus. The translation is straightforward in most cases.

$$
\begin{array}{rcl}
[\![\mathbf{NIL}]\!](c) & \stackrel{\text{def}}{=} & List(c) \\
[\![name]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)(List(l) \mid \overline{l}(\texttt{ADD}, name).\overline{l}(\texttt{RELINK}, c).\mathbf{0}) \\
[\![..]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)(List(l) \mid \overline{l}(\texttt{ADD}, \texttt{CONT}).\overline{l}(\texttt{RELINK}, c).\mathbf{0}) \\
[\![.]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)(List(l) \mid \overline{l}(\texttt{ADD}, \texttt{SELF}).\overline{l}(\texttt{RELINK}, c).\mathbf{0}) \\
[\![\_]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)(List(l) \mid \overline{l}(\texttt{ADD}, \texttt{UNIV}).\overline{l}(\texttt{RELINK}, c).\mathbf{0}) \\
[\![\&]\!](c) & \stackrel{\text{def}}{=} & Copy(self, c)
\end{array}
$$

Figure 3.8: Elements of Paths

We start with the definition of the meaning of elements of paths (Figure 3.8). There doesn't exist a separate data type for elements, so the constructs are converted straight into one-element lists. The only exception is the global self reference and the empty list. It should be stated again that the semantic function with these arguments is invoked in a context. The higher-order equation for *Block* provides that context. Hence any names of channels (like $self$) appearing in the definitions are not undefined, as it may seem first, but introduced in the *Block* equation. Alternatively we could have passed additional arguments when invoking the semantic function, but the resulting equations would have been more obscure.

Using the operations defined for lists we can translate the operations on paths in a rather straightforward way(Figure 3.9).

Sub-objects with fresh names can be created. Objects can be tested for existence, evaluated or the associated block can be retrieved. A block is accessed via a link. If a context is sent along this link the block is evaluated. A list can be converted to a block. When evaluated such a block just returns that list and ignores any context information (Figure 3.10).

Finally we define the meaning of the only four statements of the language together with the meaning of the operator for creating a block (Figure 3.11).

The top-level object (often named `Universe`) requires special treatment it has to be created automatically and contain the whole program text as associated block (Figure 3.12). The global link UNIVERSE is created in that context. The declaration of the other global (constant) links (e.g. T, F) has been omitted. After the object has been created and the program assigned to it as its associated block the object is evaluated. The returned result is volatile.

$$
\begin{array}{rcl}
[\![\textbf{head}\ r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ t,l,q)([\![r]\!](t)\ \mid\ List(l)\ \mid\ \overline{t}(\texttt{EMPTY},q).\\
& & q(v).v:[\texttt{T}\Rightarrow \overline{l}(\texttt{RELINK},c).\mathbf{0}\ \mid\ \overline{t}(\texttt{DESTROY},\texttt{NIL}).\mathbf{0},\\
& & \qquad \texttt{F}\Rightarrow \overline{t}(\texttt{HEAD},q).q(name).(\overline{l}(\texttt{ADD},name).\overline{l}(\texttt{RELINK},c).\mathbf{0}\ \mid\ \overline{t}(\texttt{DESTROY},\texttt{NIL}).\mathbf{0})])\\
[\![\textbf{last}\ r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ t,l,s,q)([\![r]\!](s)\ \mid\ Reverse(t,s)\ \mid\ \overline{s}(\texttt{EMPTY},q).\\
& & q(v).v:[\texttt{T}\Rightarrow \overline{s}(\texttt{RELINK},c).\mathbf{0},\\
& & \qquad \texttt{F}\Rightarrow List(l)\ \mid\ \overline{s}(\texttt{HEAD},q).q(name).\\
& & \qquad\quad (\overline{l}(\texttt{ADD},name).\overline{l}(\texttt{RELINK},c).\mathbf{0}\ \mid\ \overline{s}(\texttt{DESTROY},\texttt{NIL}).\mathbf{0})])\\
[\![\textbf{front}\ r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ t,l,s,q)([\![r]\!](t)\ \mid\ Reverse(t,s)\ \mid\ \overline{s}(\texttt{EMPTY},q).\\
& & q(v).v:[\texttt{T}\Rightarrow \overline{s}(\texttt{RELINK},c).\mathbf{0},\\
& & \qquad \texttt{F}\Rightarrow \overline{s}(\texttt{HEAD},q).q(name).Reverse(s,l)\ \mid\ \overline{l}(\texttt{RELINK},c).\mathbf{0}])\\
[\![\textbf{tail}\ r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ t,q)([\![r]\!](t)\ \mid\ \overline{t}(\texttt{EMPTY},q).\\
& & q(v).v:[\texttt{T}\Rightarrow \overline{t}(\texttt{RELINK},c).\mathbf{0}\\
& & \qquad \texttt{F}\Rightarrow \overline{t}(\texttt{HEAD},q).q(name).\overline{t}(\texttt{RELINK},c).\mathbf{0}])\\
[\![r_1/r_2]\!](c) & \stackrel{\text{def}}{=} & (\nu\ t_1,t_2,s)([\![r_1]\!](t_1)\ \mid\ [\![r_2]\!](t_2)\ \mid\ Reverse(t_1,s)\ \mid\ Reverse(s,t_2,c))\\
[\![r_0{=}{=}r_1]\!](c) & \stackrel{\text{def}}{=} & (\nu\ s,t,r,q)([\![r_0]\!](s)\ \mid\ [\![r_1]\!](t)\ \mid\ Copy(s,q)\ \mid\ Equal(q,t,r)\ \mid\\
& & (\nu\ a,b)(\overline{q}(\texttt{EMPTY},a).\overline{q}(\texttt{DESTROY}).\mathbf{0}\ \mid\\
& & \overline{t}(\texttt{EMPTY},b).\overline{t}(\texttt{DESTROY}).\mathbf{0}\ \mid\ \overline{r}(\texttt{DESTROY}).\mathbf{0}\ \mid\\
& & a(u).a(v).u:[\texttt{T}\Rightarrow v:[\texttt{T}\Rightarrow \overline{s}(\texttt{RELINK},c).\mathbf{0},\\
& & \qquad\qquad\qquad \texttt{F}\Rightarrow \overline{s}(\texttt{DESTROY,NIL}).List(c)],\\
& & \qquad\quad \texttt{F}\Rightarrow \overline{s}(\texttt{DESTROY},\texttt{NIL}).List(c)]))\\
[\![r_0?r_1{:}r_2]\!](c) & \stackrel{\text{def}}{=} & (\nu\ s,q,a)([\![r_0]\!](s)\ \mid\ \overline{s}(\texttt{EMPTY,a}).a(v).\\
& & v:[\texttt{T}\Rightarrow [\![r_2]\!](q)\ \mid\ \overline{q}(\texttt{RELINK},c).\mathbf{0},\\
& & \quad \texttt{F}\Rightarrow [\![r_1]\!](q)\ \mid\ \overline{q}(\texttt{RELINK},c).\mathbf{0}])
\end{array}
$$

Figure 3.9: Operations on Paths

$$
\begin{array}{rcl}
[\![\textbf{new}\ r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)([\![r]\!](l)\ \mid\ \overline{sf}(l,\texttt{NEW},c).\mathbf{0})\\
[\![\textbf{exist}\ r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)([\![r]\!](l)\ \mid\ \overline{sf}(l,\texttt{EXIST},c).\mathbf{0})\\
[\![{*}r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)([\![r]\!](l)\ \mid\ \overline{sf}(l,\texttt{EVAL},c).\mathbf{0})\\
[\![\#r]\!](c) & \stackrel{\text{def}}{=} & (\nu\ l)([\![r]\!](l)\ \mid\ \overline{sf}(l,\texttt{GET},c).\mathbf{0})\\
[\![@r]\!](c) & \stackrel{\text{def}}{=} & Block(c,(\nu\ l)([\![r]\!](l)\ \mid\ \overline{l}(\texttt{RELINK},reply).\mathbf{0})
\end{array}
$$

Figure 3.10: Accessing Objects

$$
\begin{array}{rcl}
[\![[statements]]\!](c) & \stackrel{\text{def}}{=} & Block(c,[\![statements]\!])\\
[\![{+}r;P]\!] & \stackrel{\text{def}}{=} & (\nu\ l)([\![r]\!](l)\ \mid\ \overline{sf}(l,\texttt{CREATE},\texttt{NIL}).[\![P]\!])\\
[\![{-}r;P]\!] & \stackrel{\text{def}}{=} & (\nu\ l)([\![r]\!](l)\ \mid\ \overline{sf}(l,\texttt{DESTROY},\texttt{NIL}).[\![P]\!])\\
[\![r{=}block;P]\!] & \stackrel{\text{def}}{=} & (\nu\ l,b)([\![r]\!](l)\ \mid\ [\![block]\!](b)\ \mid\ \overline{sf}(l,\texttt{SET},b).[\![P]\!])\\
[\![!r;P]\!] & \stackrel{\text{def}}{=} & (\nu\ l)([\![r]\!](l)\ \mid\ \overline{l}(\texttt{RELINK},reply).[\![P]\!])
\end{array}
$$

Figure 3.11: Statements

$$
\begin{array}{rcl}
[\![program]\!] & \stackrel{\text{def}}{=} & (\nu\ l,\texttt{UNIVERSE},b)(List(l)\ \mid\ Obj(\texttt{UNIV},\texttt{UNIVERSE},l,\texttt{UNIVERSE})\ \mid\\
& & [\![program]\!](b)\ \mid\ \overline{u}(\texttt{SET},b).\overline{u}(\texttt{EVAL},r).\mathbf{0})
\end{array}
$$

Figure 3.12: The *Universe* Object

Chapter 4

Implementation

Viewing a program or system as a mathematical object [Som89] and applying formal methods and proofs for reasoning about it is a process that is subject to human error. Only an implementation of an actual system can demonstrate whether our design achieves the intended results. However, in order to replace formal methods the testing of the system would have to be exhaustive - an impossible task in any larger development. The implementation is thus mainly a means of detecting serious (and hence during execution obvious) flaws in the design. Such feedback can then be used to improve the design further. Alternatively the occuring problems can be fixed in the implementation itself and the new design is obtained by reverse engineering. This method should only be employed for fixing minor bugs with a small overall effect.[Bei90]

4.1 General Concepts

The aim of the implementation was to demonstrate that the object-model and the language are feasible and yield useful results. It must be pointed out at this occasion that the implementation, for the mentioned reasons, only constitutes a minor part of the project. The main focus of the work was the design of the model and the language. It has to be emphasised that the implementation strategies were tuned to the specific purpose - getting feedback for the design of model and language as soon as possible. The design stage of the implementation took up most of the time. We wanted to have a design that is flexible and suitable enough for

- implementing it quickly
- extending it to a full-feature implementation

Consequently the actual testing of the implementation was reduced to a minimum and incorporated into the application of the implementation for the purposes mentioned above.

There are two aspects to the implementation (Figure 4.1) - the construction of a system based on the new object-model and the construction of translators generating programs (from programs written in the behaviour-script language) that are executable on such a system. The last stage of the process is determined by the target system and thus the structure of the incoming data also depends on it. Therefore we need different translators for different target systems. The run-time system definition is the target-system-dependent representation of the specification of the object-oriented system. All the communication and interaction between objects, their general behaviour, their creation and destruction, distribution across multi-processor systems etc. is specified in that definition. From the conceptual point of view it is thus an instantiation of the system specification.

4.1.1 The Generation of the Translator Stage

The generation of the translators follows a certain pattern (Figure 4.2). The constant *syntax definition* of the language is processed by a program that generates code for a parser. On a Unix

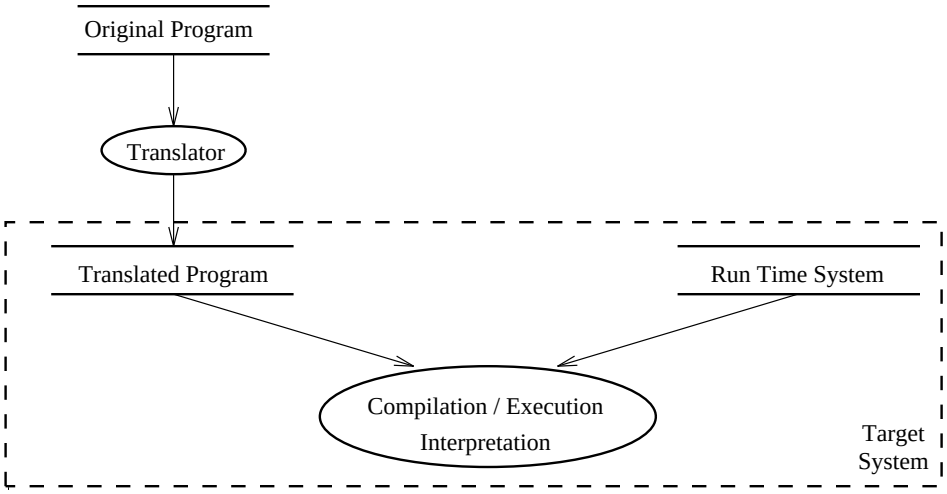

Figure 4.1: General Concept for the Implementation of the Language

platform this task is usually carried out by *lex* (cf. [Les75, Ben90]) or one of its equivalents. The *generic semantics definition* is a representation of those aspects of the language semantics that are independent from any particular implementation. In general these definitions include everything that is needed to built a parse tree of programs. The code generation stage is attached to the parse tree. This part differs from target system to target system. In the end both definitions (the parser code and the specific semantics) are processed (on a Unix platform using *yacc* (cf. [Joh75, Ben90]) or one of its equivalents) to generate the code for the translator program which is then compiled and linked to yield the actual translator.

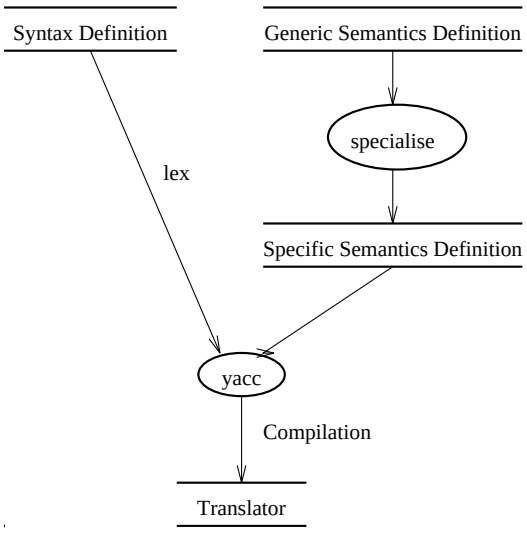

Figure 4.2: Generation of Translators for the Language

4.1.2 The Parse Tree Generation

The parse tree is an internal representation of the parsed program that is being used as input to the code generator to finally generate the code. As such it doesn't have to be a tree. For the generic semantics we chose a representation of the parse tree that closely matches the structure of the parsed program and used C++ as the language for implementing the parse tree and the

code generator. Each syntactic category of the syntax definition is assigned a C++ class and the program is represented in terms of instances of these classes. The instances of classes representing non-terminal categories include links to the instances of the contained categories, e.g. objects representing a list of statements include pointers to objects representing statements. Thus we indeed translate the program into a tree structure - a tree representing the syntactic decomposition of the program. What is passed on to the code generator is in the end a pointer to a single *program* object.

For the envisaged purposes the code generation stage could directly follow the parse tree structure to generate the code in a single pass. This is possible because our language doesn't have global identifiers. The representation allows for multiple passes though. In the simple cases the code generation is done by a member function that is attached to each class. This function calls the member function of the sub-objects (i.e. those objects representing components of the syntactic category). The results (i.e. the code associated with the sub-objects) are then combined to produce the code associated with the object. It should be noted that the combination operation differs from class to class. Hence the member functions are different. The polymorphic features of C++ are employed for that purpose. An instantiation of the generic semantics is just the attaching of different program code to the code generation member functions of the various classes.

Although the code generator is essentially a single pass compiler the *Blocks* in the language require special treatment. Definitions of blocks are nested in the language. The complete program is just one single block. For the envisaged purposes it was desirable to flatten the nested blocks and use block identifiers instead. We can still use the single pass approach as each identifier only occurs exactly once in the program - at the place were the original block definition used to be. The translated definitions themselves are collected and output separately. In fact, as the whole program itself is a block, most of the generated code comes from this stage of the compilation while the earlier traversion of the parse tree only produces some general code framework.

4.2 The π-calculus Implementation

The Implementation of the language in π-calculus (4.3) is an instance of the general implementation framework as it was presented in Figure 4.1. The key component is the translator as the actual interpretation was to be carried out with an existing program. Thus most of the work follows the general framework for the generation of translators for the language (cf. Figure 4.2). The

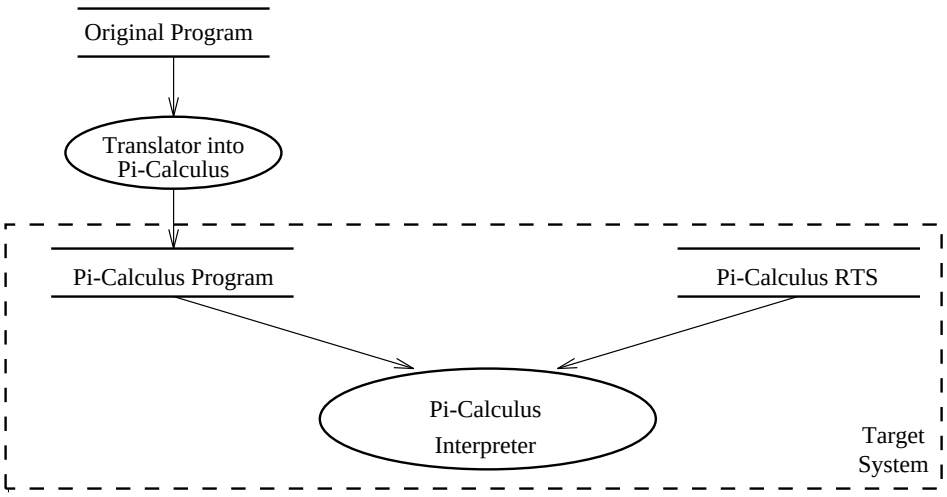

Figure 4.3: Implementation of the Language in π-calculus

implementation of the translator proved to be rather straightforward for the following reasons:

- The semantics of the language had been described in terms of the π-calculus. This in fact means that the whole translation task was completely formally specified - essentially what the translator was supposed to do is to implement the semantic function as it is specified in Section 3.2.
- The structure of the implementation (i.e. the class hierarchy) matches the structure of the definition of the semantic function very well. In both cases a compositional approach was taken. As a result there is almost a one-to-one mapping, i.e. every defining equation of the semantic function corresponds to a code generation function in the implementation.
- A similar translator (from Darwin to π-calculus) had been implemented by the author in course of related research[RE94b].

The output format of the code generator is an ASCII representation of the π-calculus. Special care was taken to find adequate renderings of the operators in order to make the resulting output intelligible for someone familiar with the usual (mathematical) representation of the calculus as it was introduced by R.Milner and as it has been used throughout this paper. As the π-calculus code was to be processed by various tools the whole generation of the output stream from the terminal elements of the π-calculus was placed into a single class (Figure C.1). Thus changes in the format of representation can be carried out easily and the translator can be adapted to different interpreters and tools for the π-calculus. An example of the output generated by the translator is given in Figure C.3. It's an excerpt of the translation of the primes sieve example program in Figure 3.1. It has to be pointed out that the current format of the output, although the rendering of the π-calculus constructs is intuitive, was designed to be readable by other programs rather than by humans. Hence the formatting is rather limited. It is rather doubtful whether a human reader could comprehend the meaning of the translation of non-trivial programs no matter how sophisticated the formatting is, although this wasn't expected anyway.

Figure C.2 is an excerpt of the class hierarchy of the translator classes. For example *BlockStatements* is a list of statements defining a block (according to the BNF of the language). Consequently the class *BlockStatements* is derived from *Block* and includes a reference to the list of statements. The member function *toPi* implements the code generator for *BlockStatements*. It is a virtual function and has been redefined to override the definition in the *Block* class.

Unfortunately it was impossible to find a program that executes or analyses the full π-calculus, including the match and mismatch operators. A substantial amount of time was spent installing, testing and investigating various tools such as the *Concurrency Work Bench CWB* [CPS89], *Imperial College Logic Environment ICLE* [Daw92], *PICT* [PRT94]. Some of them appear flexible enough for an extension. To do this would have required considerable additional effort which would only have been remotely related the project itself. Hence, for now, the π-calculus implementation remains incomplete.

4.3 The Darwin/Regis Implementation

Like the π-calculus implementation the Darwin/Regis implementation (Figure 4.4) is an instance of the general framework in Figure 4.1. Regis is a system that allows the distribution of parts of application across a network and organises the communication and data transformation. It is implemented as a C++ class hierarchy. The application is split into several processes. These processes are specified as sub-classes of a special class. Regis primitives provide means for synchronous and asynchronous communication and selective receipt. The communication interface of a process is specified in terms of typed *ports* (for incoming messages) and *port references* (for outgoing messages), both of which can be created dynamically. Thus dynamic configurations are possible. Darwin provides a higher-level approach to configurations by associating processes with components and specifying their connections. While the domain of Regis processes is flat, Darwin components can be decomposed. The flattening is done by the Darwin compiler. Darwin enables us also to allocate processors (i.e. machines) to components thus determining the pattern of distribution. A more detailed introduction to Darwin can be found in Appendix A.3.

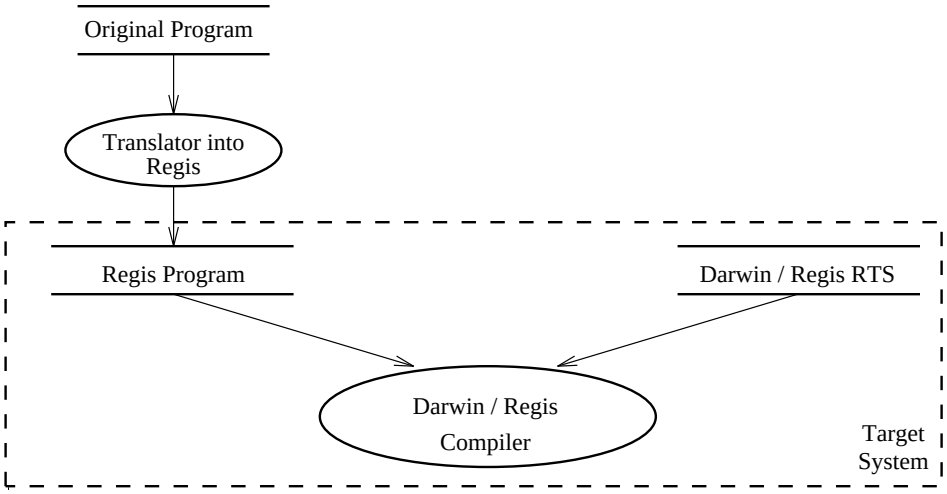

Figure 4.4: Implementation of the Language in Darwin / Regis

4.3.1 Darwin Components

The decompositional features of Darwin enable us to map the decomposition hierarchy of the system to an isomorphic structure of Darwin components. The sub-components of an *Object* component are the following (Figure 4.5):

1. *Sub-object* components. These are the components representing the sub-objects of the object in the decomposition hierarchy.

2. *ObjProc* component. This is a trivial component (i.e. it has no sub-components) and the attached process implements the features of the object model.

3. *Block* component. This trivial component represent the behaviour script that is associated with the object.

There are several reasons that justify the splitting up of the implementation of the object's behaviour (i.e. the behaviour as described in the model, not to be confused with the behaviour script) into two parts. Modelling the block as a separate object makes the actual representation and implementation of behaviour script and its operations transparent to the rest of the object. We also achieve a separation of the more managerial aspects (e.g.. the creation and destruction of objects, copying, moving, message forwarding) from the realisation of the higher-order behaviour (i.e. the behaviour as defined in the behaviour script). Furthermore the split increases the potential degree of parallelism. The only things that can potentially run in parallel in a Regis system (from a user's point of view) are the processes attached to the Darwin components. As a component has at most one attached process this limits the degree of parallelism. By having two trivial sub-components inside the *Object* component those can operate concurrently. Thus, for instance, the object can still forward messages to other objects while evaluating its behaviour script. Note also the structural isomorphism between the formal semantics and the above approach. While *ObjProc* implements the functionality specified in Figure 3.6 the definitions from Figure 3.5 are implemented by the *Block* component. The Darwin definition of the *Object* component can be found in Figure C.4 and the definitions for the *ObjProc* and *Block* components are presented in Figures C.5 and C.6 respectively.

The *Block* component is linked to the *ObjProc* component via a single connection. Along this link the *ObjProc* passes commands to the block. Such commands include the setting and retrieval of the behaviour script and its evaluation. In all cases the block interacts directly with the originator of the commands afterwards. So, for example, if the script is being evaluated the result is directly communicated to the *Block* component that issued the command. This is possible

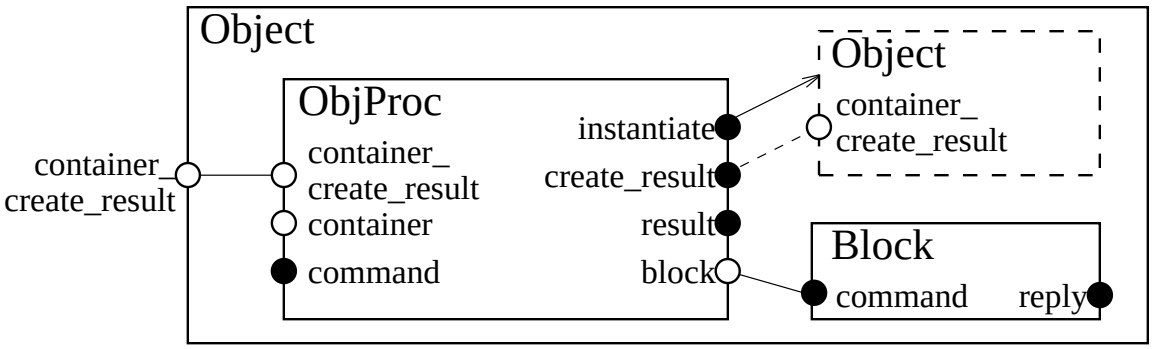

Figure 4.5: Darwin Components

because the *eval* command includes parameters that are port references to the originator. The *reply* port of a block serves the purpose of receiving the replies to messages sent to other objects.

According to the object model each object has a link to its container. This link is established via the *container* port reference of *ObjProc*. An object receives messages from a single *command* port. Sub-objects are created dynamically using the dynamic instantiation and binding features of Darwin/Regis. A new sub-object is created by sending a message to the *instantiate* port reference and dynamically re-binding the *create_result* port to the *container_create_result* port reference of the newly created component. The new object then transmits a port reference to its command channel along this link. This reference is stored in an *sub-object set* inside *ObjProc* together with the name of the new object. This set is implemented as a class *SubObjSet* that exhibits a behaviour as specified in Figure 3.4. Whenever a sub-object is accessed its command port is determined by consulting that set. The instantiation of a new sub-object is being completed by sending a port reference to the own command channel to the new object which it then binds to its *container* port reference. The *ObjProc* component has also a *result* port which is used for receiving the results of management operations on sub-objects and the block.

4.3.2 Importing Behaviour Scripts

The code that is generated by the translator must be imported by the Darwin/Regis program before it is compiled to produce an executable application. The place where this happens is the *evaluate* member function of the *Block* component. This function is called whenever a request for evaluation is transmitted to the block. As the entire translated program is included each *Block* component can potentially evaluate any behaviour script defined in the program. With this approach we avoid the transfer of segments of code between components which is extremely inefficient and causes compatibility problems in inhomogeneous distributed systems. Instead we only transfer *block identifiers*, i.e. setting and retrieving the behaviour scripts of objects only sets/retrieves associated block identifiers. Thus a *Block* component decides on which translated behaviour script to evaluate by consulting the block identifier.

When a block is evaluated an instance of an *Evaluator* class is created. For every construct of the language this class has a corresponding member function which implements the desired behaviour of that construct by a combination of operations on paths and sending commands to objects. It is thus most closely related to semantic functions from Figures 3.9, 3.10 and 3.11. Paths and the operations on them (cf. Figures 3.2, 3.3 and 3.9) are realised by a *Path* class which is used in many places throughout the implementation. This is due to the dominant role references (which are translated into paths) play in the model. The *Evaluator* class relies on the *Path* class for the implementation of the language constructs corresponding to operations on paths.

Rather than basing paths on strings we chose to use integers for the elements. The only reason for doing this is efficiency. Because of their dominant role paths are being communicated between and used by processes at a very high rate. Making these activities faster by choosing an efficient

representation of paths yields a considerable improvement in the performance. In fact the system would probably be completely useless for any real applications otherwise. The choice of the data type for the path elements is actually made at a single place in the code. Thus any improvements in this critical part of the code can be carried out easily.

4.3.3 The Translator

Like the implementation of the translator into the π-calculus the implementation of the translator into Darwin/Regis follows the general framework for the generation of translators for the language (cf. Figure 4.2). We use exactly the same classes as in the π-calculus version. The only difference is the code generator stage. There we replace the *toPi* member function of the translator classes (cf. C.2) by a *toRegis* member function. These functions have the same behavioural characteristics as there predecessors - they compose the output of the translator by invoking the *toRegis* member functions of the sub-objects and combining the result.

As in the case of the translator into π-calculus the block structure of the original program is being flattened. In fact each block is translated into a single huge C++ assignment statement. On the right-hand side of this statement is a C++ expression. The sequentialisation of statements in the language is mapped to the comma-operator in C++. All other expressions are mapped to C++ expressions representing calls to member functions of the *evaluator* object that is created when a block is evaluated. Thus the code is almost completely independent from actual semantics. This is a change to the π-calculus implementation where the code generation stage was very closely related to the semantics of the language. The Darwin/Regis translator is affected very little by changes in the semantics and completely unaffected by fine-grain modifications below the level specified in the semantics of the language. Such modifications are carried out entirely in the actual code for the system. As a further consequence the translator has no separate code generator class for generating the actual output stream. An example of the code produced by the translator can be found in Figure C.7. As the output is to be processed by the C++ compiler only very little text formatting is done.

As the current implementation uses integers to represent the elements of a path the translator has to map strings (i.e. the path elements from the program) to integers. This is done by storing all names the parser comes across in an indexed set. The code generator then uses the index associated with any particular name for computing the its corresponding integer representation. This way we ensure that re-occuring names are mapped to the same integer and that the mapping is one-to-one.

4.4 A Universal Translator

The similarity of the two translator programs shouldn't come as a surprise to the reader. After all they were deliberately designed that way by deriving them from a general framework (cf. Figure 4.2). The only part where they differ is the code generation stage and even there the data-flow is almost identical. It therefore makes sense to integrate the two translators into one single program. This turned out to be a rather straightforward process. Basically, instead of having just one code generation function for each translator class (i.e. either *toPi* or *toRegis*) we equip the classes with both of them. An excerpt of the resulting class definitions can be found in Figure C.8.

The control program can now decide which code to generate after the parsing stage. It may even choose to generate both translation without having to parse the source again. The current version of the universal translator has the following command line syntax:

```
DOOL [-t PiCalc|Regis]
```

The optional argument is a switch that decides which code is to be generated. The default is the π-calculus representation. The source code is taken from the standard input and the output is produced on the standard output. Information about the progress of the translation is sent to the standard error stream.

Chapter 5

Conclusions

5.1 Summary and Future Work

The aim of the project - to design and implement a concurrent object-oriented language for distributed programming - has been achieved. A model for objects has been devised. Based on this model a language was designed. Two translators for the language were developed. A system for executing the language in a distributed environment was built. The ultimate aim is to have

- an object-oriented language for writing applications on distributed systems.
- a complete set of tools for translating, executing, testing, analysing programs written in the language.
- a methodology of object-oriented design that is tailored for the new object-model.

There is still a long way to go to achieve these goals.

5.1.1 The Object Model

It has been shown that by investigating the very nature of objects and object-oriented systems we can develop a model which is compact and simple. Yet it includes everything that is needed. By paying attention particularly to the concurrency aspects and requirements arising from distributed systems it was ensured that the model provides all means for building an object-oriented distributed system with a high grade of concurrency.

The author believes that the developed object model in its simplicity and clarity will lessen the confusion in this field of research. Some key problems of the design of object-oriented systems either no longer arise or are overcome in an elegant way.

Valuable feedback in the model development was obtained as a result of the implementation of an actual system based on the model. As a result several changes were made in the model. The communication activities between objects now have a common pattern. Several functionally separate entities were identified, e.g. forwarding, management, processing. Finding an adequate way of destroying objects posed a particular challenge.

Future Work

The formal specification of the object model needs some improvements to adequately represent all aspects of the model. The reason why this was not done is that using the current format the length of the specification would increase considerably and become unintelligible. Either a different framework has to be found or a higher level of abstraction has to be adopted. Further research will be carried out to represent high-level object-oriented features in both the model and the language. The impact of changes in the model and the language on these features will be investigated.

For the implementation of the system in a distributed environment it is necessary to model the distribution strategy for the objects in the system. This is not a necessarily a part of the general object model as the distribution aspect is transparent. However, a model for use in an efficient implementation will have to take into account those features of the object model which enable us to dynamically distribute the objects in a way that minimises the inter-machine communication. Most probably such a model will be based on the hierarchical nature of the object-model.

5.1.2 The Language

The developed language has the potential to provide a basis for the implementation of a variety of object-oriented languages. The strength of the basic language is the way concurrency is dealt with. We hope that it is a step towards the development of a "really nice" concurrent object-oriented language.

The semantic description in terms of the π-calculus allows reasoning about the concurrency aspects of the language within a well- defined mathematical framework. Last but not least the extension of the π-calculus should lead to further discussions about the theoretical aspects of concurrency. The newly introduced *mismatch* operator extends the expressiveness of the calculus and makes it more suitable for a whole class of problems.

The feedback from the implementation of translators for the language resulted in various changes. The syntax is now more intuitive and follows some general patterns.

Future Work

The syntax of the language is not ideal and the meaning of programs written in the language is hard to comprehend. However, there has already been a significant improvement to earlier versions of the language. Also the links between the model and the language have to be expressed more clearly.

The semantic description of the language, like the formal specification of the model, does not exactly represent the intended meaning. Here again the introduction of several layers of abstraction in the description might be necessary. The new HOπ-calculus [San93a] looks like a promising candidate for that. Some optimisation should be possible to get a more compact definition with an even higher degree of concurrency.

Last but not least we have to come up with a name for the language.

5.1.3 The Implementation

The implementation of a translator of the language into the π-calculus offers the opportunity to formally analyse programs within the context of a well-defined mathematical context. Existing tools might be employed for that purpose. As the semantics of the language had been specified in terms of the π-calculus the implementation of the translator proved to be a rather straightforward process.

The implementation of a complete object-oriented system on a distributed system showed the feasibility of the approach that was taken when designing the object model and also provided valuable feedback. As a result several minor problems (like a conceptual mistake regarding the destruction of objects) in the design were uncovered and subsequently solved. The model as it has been presented in this paper thus can be demonstrated to work. Together with the translator that translates the language into Darwin/Regis (the implementation languages of the system), this represents an implementation of the language. Again we could demonstrate the feasibility of our approach (this time of the language design) and minor changes to the language were made as a result of the feedback gained from the implementation.

Future Work

A suitable system for executing π-calculus has to be found to follow the translator stage in the π-calculus implementation of the language. Several such systems have been investigated but so

far none of them handled the most recent versions of the calculus which include the *match* and *mismatch* operators. It might thus be necessary to extend and adapt one of these systems for our purposes.

To generate translators for the language for practical purposes (and not just for testing and design verification) the current implementations of the translators into π-calculus and Darwin / Regis have to be made more user-friendly and an extensive testing must be carried out. The implementation of the system itself can be modified to make it more efficient. Also a suitable strategy for automatic distribution has to be found and implemented - in the current version the system is still running on a single machine. In general the issue of distribution has to be investigated in more detail. The specification of the system deliberately doesn't cover this topic in order to make this issue transparent to the programmer.

If the language is to be used for implementing real applications a object library has to be developed. The language doesn't have any primitive data-types. These therefor have to be modelled using the existing language constructs. We also need to provide means for accessing operating system functions like i/o, interrupt handling etc. As these functions are not accessible from within the language we need to encapsulate them in special objects that are defined by the system. Although the particular implementation of these objects differs between different systems the interface that is provided in terms of the set of objects remains constant. Programs in the language that use these objects thus can still be exchanged between different systems.

Appendix A

Background

A.1 The π-calculus

The π-calculus is an elementary calculus for describing and analysing concurrent systems with evolving communication structure[MPW89, Wal91, Mil91]. The following is a brief description of the version of the calculus that has been used throughout this paper.

A system is a collection of independent *processes* which may be linked to other processes. Links have *names*; the name is the most primitive entity in the π-calculus; names have no structure. There are an infinite number of names, represented using lower-case letters. Processes *e.g.* $P, Q, R \ldots$ are built from names as follows:

- the parallel process $P \mid Q$ will execute *both* concurrently. The operation is commutative and associative.
- the replication $!P$ provides any number of copies of P. It satisfies the equation $!P = P \mid !P$

  Recursion can be recoded as replication and so need not be explicitly included as a separate method for building processes. Recursion will be used when it makes examples clearer.
- $(\nu\ y)P$ introduces a new name y with scope P. As usual, all free occurrences of y in P are bound by the quantifier, and can be uniformly renamed to any new name without changing the value of the process. The quantifier also satisfies the axioms $(\nu\ x)(\nu\ y)P = (\nu\ y)(\nu\ x)P$ provided x and y are distinct names, and $((\nu\ x)P) \mid Q = (\nu\ x)(P \mid Q)$ provided x does not occur free in Q.
- A communicating process C of the following kinds:
  - the sum $C_1 + C_2$ will execute *either* C_1 *or* C_2. The operation is commutative and associative.
  - $\overline{x}(y_1, \ldots, y_n).P$ means output the names $y_1, \ldots, y_n$ along the link x and then execute P.
  - $x(z_1, \ldots, z_n).P$ receives names $y_1, \ldots, y_n$ along x and then executes P with y_i substituted for each free occurrence of z_i in P.
  - $[x = y]P$ behaves like P if x and y are identical and like $\mathbf{0}$ otherwise.
  - $\mathbf{0}$ stops. It is an identity for both $\mid$ and $+$.

As well as being able to define processes the π-calculus defines a reduction relation between relations, written $P \rightarrow P'$, for process expressions P and P'. There is only one reduction axiom, called COMM:

$$(\cdots + x(y_1, \ldots, y_n).P \cdots) \mid (\cdots + \overline{x}(z_1, \ldots, z_n).Q \cdots) \rightarrow P\{z_1/y_1, \ldots, z_n/y_n\} \mid Q$$

Sub-processes under | and ν, but not replication or communication, may also be reduced in this way.

A.2 Extending the π-calculus with a Test for Non-Identity

Using the π-calculus for modelling *Abstract Data Types* one often encounters the problem of testing two port names for identity and then executing some action depending on the result. The original calculus defines an operator which does the testing but unfortunately is asymmetric; it does perform an action if the port names are identical, but behaves like **0** otherwise. This prevents us from using it for implementing if-then-else-decisions. The expressive power of the calculus suffers from this as it is for instance impossible to specify sets of port-names. Hence an extension of the calculus seems to be reasonable, provided that such an extension does fit into the semantic framework used for the definition of the original calculus.

A.2.1 Introducing the new operator

In this section we will introduce the new *mismatch operator* after we've recalled the definition of the *match operator* in the original calculus. Finally we show how this operator can be used for implementing if-then-else-decisions and case branching. The syntax for branching is extended with an *else*-case.

The match operator

The original calculus defines a *match operator*

$$[x = y]P$$

which behaves like P if x and y are identical and like **0** otherwise. Hence it's commonly used for testing the identity of two port names. The transition rule for the operator is defined as follows:

$$\frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$$

The mismatch operator

We now introduce a *mismatch operator* which will be the counterpart to the *match operator* defined above:

$$[x \neq y]P$$

It behaves like P if x and y are not identical and like **0** otherwise. The transition rule becomes slightly more complicated with an additional side condition:

$$\frac{P \xrightarrow{\alpha} P'}{[x \neq y]P \xrightarrow{\alpha} P'} \quad x \neq y$$

if-then-else

Having defined the new operator we can now model if-then-else-decisions on port names in the calculus, which previously was impossible. We can introduce an abbreviation

$$[x?y]P : Q$$

which means

$$[x = y]P + [x \neq y]Q$$

and behaves like P if x and y are identical and like Q otherwise.

Branching

It is often useful to allow input names to determine the further course of computation. For this purpose another abbreviation is introduced (cf. [MPW89]):

$$x : [y_1 \Rightarrow P_1, y_2 \Rightarrow P_2, \ldots, y_n \Rightarrow P_n]$$

meaning

$$x(v).([v = y_1]P_1 + [v = y_2]P_2 + \ldots + [v = y_n]P_n)$$

We now extend this with an additional *else* branch. As we sometimes shall wish to use the passed port name in the else branch we have to specify explicitly the input variable. We will therefore separate the actual input operation from the branching again:

$$x(v).v : [y_1 \Rightarrow P_1, y_2 \Rightarrow P_2, \ldots, y_n \Rightarrow P_n, else \Rightarrow Q]$$

This is an abbreviation for:

$$x(v).([v = y_1]P_1 + [v = y_2]P_2 + \ldots + [v = y_n]P_n + [v \neq y_1][v \neq y_2] \cdots [v \neq y_n]Q)$$

A.2.2 Example

As an example of the application of the extended calculus we try to model sets of port names. The element-based operations on sets are *add, remove, exist*, with *add* having to ensure that the element to be added is not included in the set already.

$$\begin{aligned}
Set(c) \quad &\stackrel{\text{def}}{=} \quad c(v,m,r).v : [\texttt{ADD} \Rightarrow (\nu\ tail)(Set(c, tail, m) \ \mid \ Set(tail)),\\
&\qquad\qquad \texttt{REMOVE} \Rightarrow Set(c),\\
&\qquad\qquad \texttt{EXIST} \Rightarrow \overline{r}(\texttt{F}).Set(c),\\
&\qquad\qquad \texttt{RELINK} \Rightarrow Set(r)]\\
Set(c,t,n) \quad &\stackrel{\text{def}}{=} \quad c(v,m,r).v : [\texttt{ADD} \Rightarrow m : [n \Rightarrow Set(c,t,n),\\
&\qquad\qquad\qquad\qquad else \Rightarrow \overline{t}(\texttt{ADD}, m, \texttt{NIL}).Set(c,t,n)],\\
&\qquad\qquad \texttt{REMOVE} \Rightarrow m : [n \Rightarrow \overline{t}(\texttt{RELINK}, \texttt{NIL}, c).\mathbf{0},\\
&\qquad\qquad\qquad\qquad else \Rightarrow \overline{t}(\texttt{REMOVE}, m, \texttt{NIL}).Set(c,t,n)],\\
&\qquad\qquad \texttt{EXIST} \Rightarrow m : [n \Rightarrow \overline{r}(\texttt{T}).Set(c,t,n),\\
&\qquad\qquad\qquad\qquad else \Rightarrow \overline{t}(\texttt{EXIST}, m, r).Set(c,t,n)],\\
&\qquad\qquad \texttt{RELINK} \Rightarrow Set(r,t,n)]
\end{aligned}$$

The set is implemented as a linked list. Each element of the list apart from the last one stores one port name. Each element receives messages along a special port. Only the message port of the head element is known to the environment. Initially the list consists only of the special terminating element. Elements are added just before the terminating element. If an element is removed from the set, the message port of the following element has to be changed to the message port of the removed element. This is done with the *relink* operation. Trying to remove the terminating element leaves the list as it is.

Note, that we don't use the original definition of the operators in the calculus but rather a definition introduced by Walker in [Wal91].

The basic principle of all the operations is the same. All messages from the environment are passed to the head of the list. A message consists of an operation selector and two arguments. If less arguments are required the rest is set to `NIL`.If the operation applies to this element it is performed. Otherwise the received message is sent along the message port of the next element in the list. At this point the new branching operator is used.

The terminating element has a different functionality then the other elements. Any message sent to this element means that the element it was intended for is not in the list. This is used for the *exist* operation and for the *add* to ensure that port names are only stored once in the list.

A.2.3 The Impact on the Calculus

The introduction of the *mismatch* operator affects the properties of the transition system and of the different versions of bisimulation. Davide Sangiorgi points this out in his paper on bisimulation [San93b]. The following property of the transition system doesn't hold anymore (cf. Lemma 3 in [MPW89]):

If $P \xrightarrow{\alpha} P'$ then $P\sigma \xrightarrow{\alpha\sigma} P'\sigma$

i.e. a substitution must not decrease the capabilities of an agent to perform an action. Hence substitution is monotonic with respect to the number of possible transactions. To illustrate how the *mismatch* operator violates this property let's consider the following process and substitution:

$$P \stackrel{\text{def}}{=} [x \neq y]x(z).P' \quad \text{and} \quad \sigma \stackrel{\text{def}}{=} \{x/y\}$$

According to the transition rule for *mismatch* the agent can engage in a transition $P \xrightarrow{x(z)} P'$ if $x \neq y$. If we apply the substitution we get

$$P\sigma \stackrel{\text{def}}{=} [x \neq x]x(z).P'\sigma$$

Now $P\sigma$ clearly can't perform the action as the side-condition can never be satisfied. However the monotonicity property can be restored if we restrict the substitution to be injective.

A substitution σ is injective if and only if
$\forall x, y \in N$ if $x \neq y$ then $x\sigma \neq y\sigma$.

The proof of the lemma for the *mismatch* operator is straightforward:

Let $P \stackrel{\text{def}}{=} [x \neq y]P_1$ with $x \neq y$ and $P_1 \xrightarrow{\alpha} P'$, so $P_1\sigma \xrightarrow{\alpha\sigma} P'\sigma$. Hence since $x\sigma \neq y\sigma$, $P\sigma \xrightarrow{\alpha} P'\sigma$.

The converse of the lemma now holds as well.

Unfortunately injective substitutions don't suffice our transition system in general. Consider the following example:

$$\overline{x}(z).\mathbf{0} \mid x(y).\overline{x}(z).\mathbf{0}$$

Applying the transition rule for parallel composition we get

$$\mathbf{0} \mid (\overline{y}(z).\mathbf{0})\{y/z\}$$

The occurring substitution is clearly non-injective. It can be shown that we still can maintain the notion of strong bisimilarity and late bisimilarity. Furthermore the *mismatch* operator is essential to get a complete axiomatisation of early bisimulation. On the other hand the notion of open bisimulation is destroyed by introducing the operator. For a detailed discussion we refer to the papers of Joachim Parrow and Davide Sangiorgi ([JS93, San93b]).

A.3 Darwin

The Darwin language[Dul92, MDK93, JKD92] is intended for the description of a hierarchical configurations of processes. Composite components are written in the configuration language itself and primitive components are written in C++. A grammar for the configuration language is given in Appendix A.4. The Darwin system grew out of Conic which has been used in large scale industrial problems[JMS89]. Current implementations are based on the *Regis* distributed system[MKD93]. The semantics of Darwin programs has been formally specified in terms of the π-calculus[EP93, RE94b]. An important characteristic of the Darwin system is that it enables systems to be configured dynamically by making the addresses of ports first class objects.

The basic unit of the configuration language is a component or process. A Darwin configuration component consists of a list of declarations. Instances of components are declared using the **inst** keyword. The **inst** declaration actually causes a process to be created. The interface of a component with its environment is a vector of names that are **provide**d by the process, and another vector of names that are **require**d by the process. These names may be of any type; the Darwin configuration language does not place any restrictions on the types (but it does ensure that program construction is type secure). Although the configuration language does not place any restriction on what can be a provision or requirement, in the examples used here they will refer to communication ports for unidirectional communication where the information is transmitted from the required to the provided ports. The names of provisions and requirements may be referred to in the program defining the process.

A **bind** declaration is used in order to bind two components together to enable the transference of data . This is the core statement of the configuration language.

bind $id_1.requirement - id_2.provision$

assigns the provided value of one instance to the required name of another. The **bind** statement may also refer to the provisions and requirements of the process being defined.

Darwin also enables a limited amount of control structuring of its declarations. **When** declarations are guards that enable one or more declarations to be optionally declared. Since it is not uncommon to want to perform the same declaration repetitively there are **forall** declarations.

The following is an example of a Darwin program and its graphical representation:

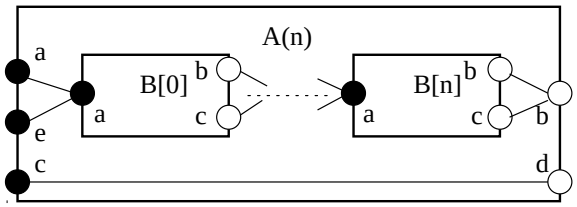

Figure A.1: Example of a Configuration

```
component B {
provide a;
require b,c;
        }
```

```
component A(int n) {
provide a, c, e;
require b, d;
array   B[n]:B;
forall i:0..n
        inst B[n];
forall i:0..n-1 {
        bind    B[i].b--B[i+1].a;
                B[i].c--B[i+1].a;
                }
bind    a--B[0].a; e--B[0].a;
        B[n].b--b; B[n].c--b;
        c--d;
        }
```

The abstract component `B` has three ports - one provided (`a`) and two required (`b,c`). The component `A` has three provided ports (`a,c,e`) and two required ports (`b,d`). Inside `A` $n+1$ instances of the abstract component `B` are chained together by connecting their required ports `b,c` to the provided port `a` of the next component in the chain. The provided port `a` of the first component is connected to the provided ports `a,e` of the composite component. Similarly the

required ports `b,c` of the last component in the chain are connected to the required port `b` of the composite component. Finally there exists a direct link between the provided port `c` and the required port `d`.

A.4 Darwin Syntax

DarwinSpec = *Component*
| *DarwinSpec Component*

Component = **component** *Id FormalParameters ComponentBody*

FormalParameters =
| (*ParameterList*)

ParameterList = *Parameter*
| *ParameterList* , *Parameter*

Parameter = *Type Id*
| *Type Id* = *Expr*
| **component** *Id*
| **component** *Id* = *Id*

Type = **int**
| **double**
| **char**
| **char***

ComponentBody =
| { *SentenceList* }

SentenceList = *Sentence*
| *SentenceList Sentence*

Sentence = ;
| **const** *ConstDeclList* ;
| **when** *Expr Block* ;
| **forall** *Id* : *Expr* .. *Expr Block* ;
| **provide** *InterfaceDeclList* ;
| **require** *InterfaceDeclList* ;
| **inst** *InstanceDeclList* ;
| **array** *InstanceList* ;
| **bind** *BindClauseList* ;
| **verbatim** { *any characters* }**$**

Block = *Sentence*
| { *SentenceList* }

ConstDeclList = *ConstDecl*
| *ConstList* ; *ConstDecl*

ConstDecl = *Type Id* = *Expr*

InterfaceDeclList = *InterfaceDecl*
| *InterfaceDeclList* , *InterfaceDecl*

InterfaceDecl = *Interface*
| *Interface TypeConstraint*

Interface = *Id*
| *Id*[*Expr*]

TypeConstraint = < *Id* >
| < *String* >
| << *Id* >>

InstanceDeclList = *InstanceDecl*
| *InstanceDeclList* ; *InstanceDecl*

InstanceDecl = *Instance Arguments Location*

Arguments =
| (*ExprList*)

ExprList = *Expr*
| *ExprList* , *Expr*

Location =
| @ *Expr*

InstanceList = *Instance*
| *InstanceList* ; *Instance*

Instance = *InstName*
| *InstName* : *Id*
| *InstName* : **dyn** *Id*

InstName = *Id*
| *Id*[*Expr*]

BindClauseList = *Binding*
| *BindClauseList* ; *Binding*

Binding = *Port* - - *Port*

Port = *Interface*
| *Id* . *Interface*

Id = (*letter* | _){*letter* | *digit* | _}

Expr = *Subset of regular C++ Expressions*

Appendix B

Object Model

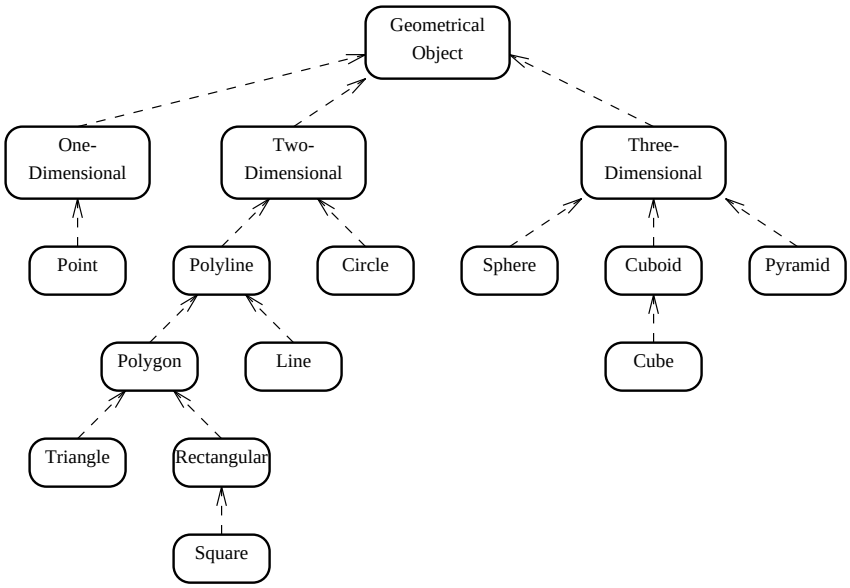

Figure B.1: Objects in an Abstraction Hierarchy

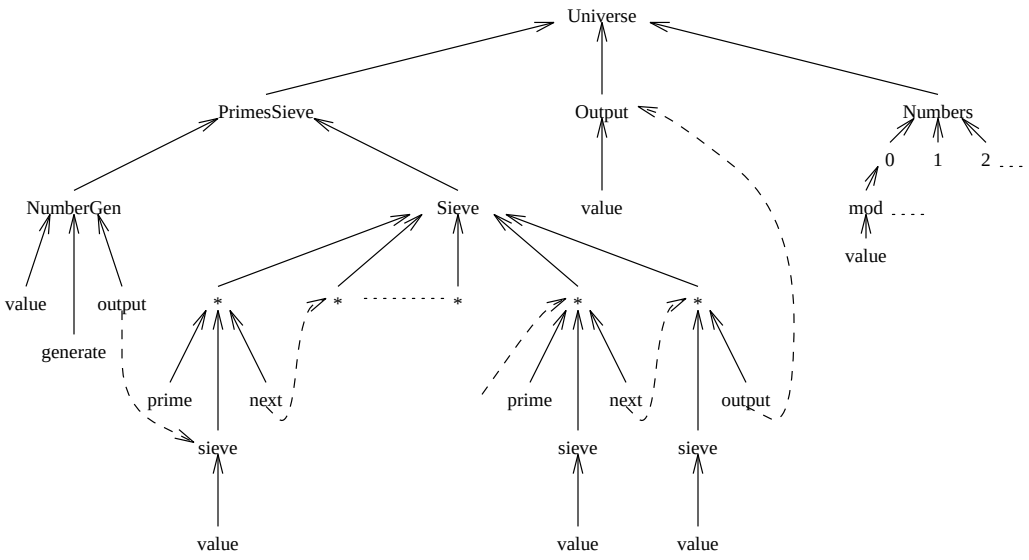

Figure B.2: Decomposition Hierarchy of the Primes Sieve Example

Appendix C

Implementation

C.1 π-calculus Implementation

```
#ifndef PICALC_H
#define PICALC_H

class PiCalc {
public:
        static const String arguments(const String &arg);
        static const String definition(const String &n, const String &arg);
        static const String input(const String &ch, const String &arg);
        static const String output(const String &ch, const String &arg);
        static const String create(const String &arg);                    10
        static const String newname(const String &arg);
        static const String par();
        };

#endif
```

Figure C.1: The π-calculus Code Generator Class

```
// This may look like C code, but it is really -- C++ --
#ifndef BLOCK_H
#define BLOCK_H

#include "ptypes.h"
#include "ParserObject.h"

class Block: public ParserObject {
  public:
      Block();                                                          10
      virtual String toPi(const String &) const;
  private:
      };

class BlockStatements: public Block {
  public:
      BlockStatements(const StatementList &);
      virtual String toPi(const String &) const;
  private:
      const   StatementList   &statements;                              20
      };

class BlockRef: public Block {
  public:
      BlockRef(const Obj &);
      virtual String toPi(const String &) const;
  private:
      const   Obj     &obj;
      };
                                                                        30
class BlockBlock: public Block {
  public:
      BlockBlock(const Obj &);
      virtual         String toPi(const String &) const;
  private:
      const   Obj     &obj;
      };

#endif
```

Figure C.2: Excerpt from the Class Hierarchy

```
(# l,UNIVERSE,b)(List(l) | Obj(UNIVERSE,UNIVERSE,l,UNIVERSE) |
(Block_21(b)) | ^u(SET,b).^u(EVAL,r).0)

Block_0(c) = !(c(u,v).u:[RELINK->Block_0(v),EVAL->v(self,cnt,sf,reply).(# l)
(((# l)(((# t_1,t_2)(((# l)(List(l) | ^l(ADD,CONT).^l(RELINK,t_1).0)) |
((# l)(List(l) | ^l(ADD,_value).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) |
^sf(l,EVAL,l).0)) | ^l(RELINK,reply).0)])

Block_1(c) = !(c(u,v).u:[RELINK->Block_1(v),EVAL->v(self,cnt,sf,reply).(# l)
(((# l)(((# l)(((# t_1,t_2)(((# l)(List(l) | ^l(ADD,CONT).^l(RELINK,t_1).0)) |
((# l)(List(l) | ^l(ADD,_output).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) |
^sf(l,EVAL,l).0)) | ^sf(l,EVAL,l).0)) | ^l(RELINK,reply).0)])

Block_2(c) = !(c(u,v).u:[RELINK->Block_2(v),EVAL->v(self,cnt,sf,reply).(# l)
(((# l)(((# l)(((# t_1,t_2)(((# t_1,t_2)(((# l)(List(l) |
^l(ADD,CONT).^l(RELINK,t_1).0)) | ((# l)(List(l) |
^l(ADD,_value).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) | ((# l)(List(l) |
^l(ADD,_inc2).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) | ^sf(l,EVAL,l).0)) |
^sf(l,EVAL,l).0)) | ^l(RELINK,reply).0)])

Block_3(c) = !(c(u,v).u:[RELINK->Block_3(v),EVAL->v(self,cnt,sf,reply).(# l)
(((# l)(((# t_1,t_2)(((# l)(List(l) | ^l(ADD,CONT).^l(RELINK,t_1).0)) | ((# l)
((Copy(self,l)) | Last(l))) | Concat(t_1,t_2))) | ^sf(l,EVAL,l).0)) |
^l(RELINK,reply).0)])

Block_4(c) = !(c(u,v).u:[RELINK->Block_4(v),EVAL->v(self,cnt,sf,reply).(# l)
((List(l)) | ^l(RELINK,reply).((# l)(((# l)(((# t_1,t_2)(((# t_1,t_2)(((# l)
(List(l) | ^l(ADD,CONT).^l(RELINK,t_1).0)) | ((# l)(List(l) |
^l(ADD,_output).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) | ((# l)(List(l) |
^l(ADD,_value).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) | ^sf(l,EVAL,l).0)) |
Block_0(b) | ^sf(l,SET,b).((# l)(((# l)(List(l) |
^l(ADD,_tmp).^l(RELINK,l).0)) | Block_1(b) | ^sf(l,SET,b).((# l)(((# t_1,t_2)
(((# l)(List(l) | ^l(ADD,CONT).^l(RELINK,t_1).0)) | ((# l)(List(l) |
^l(ADD,_value).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) | Block_2(b) |
^sf(l,SET,b).((# l)(((# l)(List(l) | ^l(ADD,_tmp).^l(RELINK,l).0)) |
Block_3(b) | ^sf(l,SET,b).0))))))))))])

Block_5(c) = !(c(u,v).u:[RELINK->Block_5(v),EVAL->v(self,cnt,sf,reply).(# l)
(((# l)(((# l)(List(l) | ^l(ADD,_Sieve).^l(RELINK,l).0)) | ^sf(l,NEW,l).0)) |
^l(RELINK,reply).0)])

Block_6(c) = !(c(u,v).u:[RELINK->Block_6(v),EVAL->v(self,cnt,sf,reply).(# l)
(((# l)(((# t_1,t_2)(((# l)(List(l) | ^l(ADD,_tmp).^l(RELINK,t_1).0)) |
((# l)(List(l) | ^l(ADD,_sieve).^l(RELINK,t_2).0)) | Concat(t_1,t_2))) |
^sf(l,EVAL,l).0)) | ^l(RELINK,reply).0)])
```

Figure C.3: Example Output of π-calculus Translator

C.2 Darwin / Regis Implementation

```
// This may look like C code, but it is really -- C++ --
#include "ObjProc.dw"
#include "Block.dw"

component Object(int un) {
        require container_create_result<port CreateResult>;
                //a portref to the command channel is reported back along this
        inst process: ObjProc(un);
        inst block: Block;
        bind process.container_create_result -- container_create_result;
        bind process.block -- block.command;
                //forwarding
        bind process.instantiate -- dyn Object;
                //initiates the creation of a sub-object
        bind dyn Object.container_create_result -- process.create_result;
                //forwarding
        }
```

Figure C.4: The *Object* Component

```
// This may look like C code, but it is really -- C++ --

component ObjProc(int un) {
      require instantiate<dyn int>;
      provide command<port Command>;
      provide create_result<port CreateResult>;
      require container_create_result<port CreateResult>;
      require container<port Command>;
      require block<port BlockCommand>;
      provide result<port BlockReply>;                                    10
#pragma verbatim
      SubObjSet      sub_objects;
      Path    self;
      void    destruct();
      Pix     findLink(CommandChannel &ret, const Name &n);
      void    executeCommand(const command &com, ReplyChannel &channel,
                      Path &arg, CommandChannel &ch);
      void    executeContainerCommand(Path &path,const command &com,
                              ReplyChannel &channel, Path &arg,
                              CommandChannel &ch);                        20
      void    inst(CommandChannel &ret, Path &p, CommandChannel &cont);
      int     createSubObj(CommandChannel &ret, const Name &n);
      void    copy(CommandChannel &ret, const CommandChannel &cont);
      void    rename(Path path);
      void    destroySubObjects();
      Pix     destroySubObj(const Name &n);
      void    eval(const ReplyChannel &r);
      void    set(const ReplyChannel &r, Path &arg);
      void    get(const ReplyChannel &r);
      int     child_count;                                                30
      int     locked;
#pragma end
      }
```

Figure C.5: The *ObjProc* Component

```
// This may look like C code, but it is really -- C++ --

component Block {
      provide command<port BlockCommand>;
      provide reply<port BlockReply>;
#pragma verbatim
      Name getUniverseBID();
      void evaluate();
      void set();
      void get();                                                         10
      Path execute(const command &c, Path &p);
      Command com;
      BlockReply result;
      BlockCommand bc;
      Path block;
#pragma end
      }
```

Figure C.6: The *Block* Component

```
case 0:
p=(e.lock(e.eval(e.concat(NAME_CONT,103))),\
(e.ret(NIL),\
(e.assignObj(e.concat(e.eval(e.concat(NAME_CONT,103)),104),\
e.concat(NAME_CONT,104)),\
(e.eval(e.eval(e.concat(NAME_CONT,103))),\
(e.unlock(e.eval(e.concat(NAME_CONT,103))),\
(e.convert(e.concat(NAME_CONT,104),\
e.eval(e.concat(e.eval(e.concat(NAME_CONT,104)),105))),\
e.eval(NAME_SELF)))))));
break;
case 1:
p=(e.lock(e.concat(e.eval(e.concat(NAME_CONT,111)),107)),\
(e.ret(NIL),\
(e.assignObj(e.concat(e.eval(e.concat(NAME_CONT,111)),e.concat(107,104)),104),\
(e.convert(e.eval(e.concat(104,e.concat(113,104))),\
e.eval(e.concat(NAME_CONT,112))),\
(e.convert(106,e.branch(e.equal(e.eval(e.concat(e.eval(104),113)),\
e.concat(108,e.concat(109,114))),e.equal(e.eval(e.concat(e.eval(104),113)),\
e.concat(108,e.concat(109,114))),\
e.concat(e.eval(e.concat(NAME_CONT,111)),107))),\
(e.eval(e.eval(106)),\
e.unlock(e.concat(e.eval(e.concat(NAME_CONT,111)),107)))))))));
break;
case 2:
p=(e.convert(e.concat(NAME_CONT,111),e.newObj(e.concat(NAME_CONT,NAME_CONT))),\
(e.assignObj(e.eval(e.concat(NAME_CONT,111)),NAME_CONT),\
(e.convert(e.concat(NAME_CONT,112),e.eval(104)),\
(e.destroy(e.concat(NAME_CONT,103)),\
(e.assignBlock(NAME_SELF,1),\
(e.lock(e.eval(e.concat(NAME_CONT,103))),\
(e.ret(NIL),\
(e.assignObj(e.concat(e.eval(e.concat(NAME_CONT,103)),104),104),\
(e.eval(e.eval(e.concat(NAME_CONT,103))),\
e.unlock(e.eval(e.concat(NAME_CONT,103))))))))))));
break;
```

Figure C.7: Example Output of Darwin/Regis Translator

C.3 Universal Translator

```
// This may look like C code, but it is really -- C++ --
#ifndef _SIMPLEREF_H
#define _SIMPLEREF_H

#include "ptypes.h"
#include "ParserObject.h"
#include <DLList.h>

class SimpleRef: public ParserObject {
public:                                                          10
        SimpleRef();
        virtual String toPi(const String &) const;
        virtual String toRegis() const;
        };

class SimpleRefName: public SimpleRef {
public:
        SimpleRefName(const String &);
        virtual String toPi(const String &) const;
        virtual String toRegis() const;                          20
private:
        static int addName(const String &);
        static int findName(const String &);
        const String &name;
        static DLList<String>   names;
        };

#endif
```

Figure C.8: The *SimpleRef* Classes of the Universal Translator

Bibliography

[Agh86] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Atk91] C. Atkinson. *Object-Oriented Reuse, Concurrency and Distribution.* Addison-Wesley, 1991.

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

[BC87] E. Blake and S. Cook. On including part hierarchies in object-oriented languages. In *ECOOP'87 Proceedings.* Springer-Verlag, 1987.

[BC92] Y. Bekkers and J. Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Berlin, 1992. Springer-Verlag.

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Rheinhold, 2nd edition, 1990.

[Ben90] J.P. Bennett. *Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC.* McGraw-Hill, 1990.

[Boo94] G. Booch. *Object-Oriented Analysis and Design.* The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings, Redwood City, California, 1994.

[BY87] J.-B. Briot and A. Yonezawa. Inheritance and synchronisation in concurrent oop. In O. Nierstrasz, editor, *ECOOP'87 Proceedings*, Lecture Notes In Computer Science. Springer-Verlag, 1987.

[Cox86] B.J. Cox. *Object-Oriented Programming : An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based verification tool for finite-state systems. In *Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Design.* Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, 1991.

[Daw92] W.M.G. Dawson. A generic logic environment. In A. Voronkov, editor, *Proceedings of Logic Programming and Automated Reasoning*, LNAI 624. Springer Verlag, 1992.

[Dul92] N. Dulay. The Darwin configuration language. Imperial College Department of Computing Internal Report, March 1992.

[EP93] S. Eisenbach and R. Patterson. π-calculus semantics for the concurrent configuration language darwin. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 2. IEEE Computer Society Press, 1993.

[Fro92] S. Frolund. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In *ECOOP'92 Proceedings*. Springer-Verlag, 1992.

[GR89] A. Goldberg and D. Robson. *SmallTalk-80: The Language*. Addison-Wesley, 1989.

[JA92] S. Jagannathan and G.A. Agha. A reflective model of inheritance. In *ECOOP'92 Proceedings*. Springer-Verlag, 1992.

[JKD92] M.S. Sloman J. Kramer, J. Magee and N. Dulay. Configuring object based distributed programs in rex. In *IEE Software Engineering Journal*, March 1992.

[JMS89] J. Kramer J. Magee and M. Sloman. Constructing distributed programs in conic. *IEEE Transactions on Software Engineering*, 15, 1989.

[Joh75] S.C. Johnson. Yacc - yet another compiler compiler. Technical report, Bell Laboratories, 1975.

[JS93] J.Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. Technical Report ECS-LFCS 93-262, University of Edinburgh, 1993.

[KL89] D. Kafura and K. Hae Lee. Inheritance in actor based concurrent object-oriented languages. In O. Nierstrasz, editor, *ECOOP'89 Proceedings*, Lecture Notes In Computer Science. Springer-Verlag, 1989.

[KL93] D. Kafura and R. Lavender. Concurrent object-oriented languages and the inheritance anomaly. In *ISIPCALA'93 Proceedings*, 1993.

[Les75] M.E. Lesk. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, 1975.

[MDK93] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, 8(2):73–82, March 1993.

[MDK94] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programming. In *2nd International Workshop on Configurable Distributed Systems*. IEEE Comp Society Press, March 1994.

[Mes93] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. Nierstrasz, editor, *ECOOP'93 Proceedings*, volume 707 of *Lecture Notes In Computer Science*. Springer Verlag, 1993.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science. Prentice-Hall, New York, 1988.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, October 1991.

[MKD93] J. Magee, J. Kramer, and N. Dulay. Darwin/mp: An environment for parallel and distributed programming. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, volume 2. IEEE Computer Society Press, 1993.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part i and ii. Technical Report ECS-LFCS 89-86/87, University of Edinburgh, 1989.

[MWY91] S. Matsuoka, T. Wanatabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In O. Nierstrasz, editor, *ECOOP'91 Proceedings*, Lecture Notes In Computer Science. Springer-Verlag, 1991.

[Neu91] C. Neusius. Synchronizing actions. In O. Nierstrasz, editor, *ECOOP'91 Proceedings*, Lecture Notes In Computer Science. Springer-Verlag, 1991.

[PB94] R. Pandey and J. Browne. A compositional approach to concurrent object-oriented programming. In *ICCL'94 Proceedings*, 1994.

[PRT94] Benjamin Pierce, Didier Rémy, and David Turner. PICT: A typed, higher-order concurrent programming language based on the π-calculus, 1994. Available by anonymous FTP from `pub/bcp` on `ftp.dcs.ed.ac.uk`.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall International, London, 1991.

[RE94a] M. Radestock and S. Eisenbach. Towards a minimal object-oriented language for distributed and concurrent programming. In *To appear in: PODC'94 Conference Proceedings.* Springer-Verlag, 1994.

[RE94b] M. Radestock and S. Eisenbach. What do you get from a π-calculus semantics? In *To appear in: PARLE'94 Conference Proceedings.* Springer-Verlag, 1994.

[RS92] M. Radestock and W. Schubert. Ein framebasiertes datenverwaltungssystem. Technical report, Technische Universitaet Dresden, November 1992.

[Rya93] M. Ryan. Specification of state-based systems. Technical report, Imperial College Department of Computing, January 1993.

[San93a] D. Sangiorgi. From π-calculus to higher-order π-calculus – and back. Technical report, University of Edinburgh, 1993.

[San93b] D. Sangiorgi. A theory of bisimulation for the π-calculus. Technical report, University of Edinburgh, May 1993.

[Sch86] A. Schiper. *Concurrent Programming.* North Oxford Academic, 1986.

[Som89] I. Sommerville. *Software Engineering.* Addison-Wesley, 3rd edition, 1989.

[W+91] P. Wegner et al. Panel: What is an object? In O. Nierstrasz, editor, *Object-Based Concurrent Computing; ECOOP'91 Workshop Proceedings*, volume 707 of *Lecture Notes In Computer Science.* Springer-Verlag, 1991.

[Wal89] D. Walker. Some results on the π-calculus. In *Concurrency: Theory, Language, and Architecture*, Oxford, UK, September 1989.

[Wal91] D. Walker. π-calculus semantics of object-oriented programming languages. In *Conference on Theoretical Aspects of Computer Software*, Tohoku University, Japan, September 1991.

[Wal93] D. Walker. Process calculus and parallel object-oriented programming languages. In *ISIPCALA'93 Proceedings*, 1993.

[WFN90] E.F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation execution in distributed systems. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, June 1990.

[YM93] A. Yonezawa and S. Matsuoka. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency.* MIT Press, 1993.

[YT87] A. Yonezawa and M.D. Tokoro. *Object-Oriented Concurrent Programming.* MIT Press, 1987.