Process- and agent-based modelling techniques for dialogue systems and virtual environments

B.W. van Schooten

Faculty of Computer Science, University of Twente

P.O. Box 217, 7500 AE, Enschede, The Netherlands

schooten@cs.utwente.nl

March 6, 2000

Abstract

This text presents results of ongoing research, which is aimed at developing a framework for developing multimodal natural language dialogue systems operating within virtual environments. The aspects of multimodality and presence in a virtual environment are chosen as the main focus of this research. It may be argued that specification techniques would form the basis of such a framework. Therefore, a general overview and evaluation is given of existing specification techniques for interactive systems, based on both literature and previous research results. This includes the object-oriented model, process algebras, interactor models, and agent systems. Agent systems are further subdivided into intentional logics, production rule systems, agent communication languages, agent platforms, and agent architectures.

A new agent system is proposed, which is based on update notification mechanisms as found in interactor models, and the 'facilitator' function as found in some agent platforms. In this system, agent-to-agent communication proceeds through buffered two-way channels. Each agent is able to manage any number of channels. In addition, the agent can query the agents it knows for information about their properties and channels, i.e. its 'environment'. This is argued to be especially useful for environmental awareness in a constantly changing virtual environment. The system is illustrated with some examples.

Each agent in this system may still be specified using an arbitrary language. It will be argued that a language based on production rules would be a natural choice.

Contents

1 Project description **3**

1.1 Foreword . . . 3

1.2 Natural interaction . . . 3

1.3 A framework for developing natural interaction . . . 4

1.4 Current state of research . . . 7

2 Process communication models **8**

2.1 The object-oriented model . . . 9

2.1.1 Discussion . . . 10

2.2 Process algebras . . . 10

2.2.1 Discussion . . . 11

2.2.2 The object-based model in process algebra . . . 13

2.3 Interactors and dataflows . . . 16

2.3.1 Discussion . . . 21

2.4 Agents . . . 23

2.4.1 Intentional logics . . . 25

2.4.2 Production rule systems . . . 26

2.4.3 Task negotiation schemes . . . 29

2.4.4 Agent platforms . . . 30

2.4.5 Agent architectures . . . 33

2.4.6 Agent communication language standards . . . 33

2.4.7 Discussion . . . 36

3 Proposal of agent system **38**

3.1 First prototype . . . 38

3.1.1 Discussion . . . 40

3.2 Conclusions and future research . . . 44

4 Appendix: a developer's guide **46**

4.1 Overview of API . . . 46

4.2 A small design guidebook . . . 48

4.2.1 Examples . . . 53

5 References **56**

1 Project description

1.1 Foreword

This text is a result of my work as a Ph.D. student at the Parlevink project, which is part of the TKI (Taal, Kennis, en Interactie) group. Thanks go to E.M.A.G. van Dijk, A. Nijholt, J. Hulstijn, R. op den Akker, J. Zwiers, M. Evers, and P.R.J. Asveld for reviewing various parts and versions of this text.

1.2 Natural interaction

As computers are becoming more widespread, people have attempted to make interaction with computer systems more useable and learnable. One of these attempts is the natural language (NL) dialogue system, which will simply be called 'dialogue system' here. A dialogue system is a system that is able to understand and mimick 'natural' human linguistic behaviour to such an extent, that humans are able to interact with it without having to learn computer-specific interaction first.

General frameworks to help in building dialogue systems have been relatively well-developed. These include methodological frameworks (Gibbon et al., 1997) (Bernsen et al., 1998), design guidelines and solution templates (Bernsen et al., 1998) (Smith, 1997), specification techniques (Philips, 1998) (Jönsson, 1993), and evaluation techniques (Walker et al., 1997) (Sparck Jones and Galliers, 1996).

It is clear by looking at these frameworks that dialogue design remains a hard problem. Attempts at dialogue systems have always implied 'hard-coding' the system with rigid behaviour, appropriate only to deal with user utterances within a very limited domain. In particular, NL parsing, knowledge modelling, and dialogue management remain hard problems for which no general theory providing a general solution exists. Hence, developing dialogue systems is a matter of iterative development and situation-specific engineering. The frameworks are engineering frameworks that help to ensure quality by pointing out problems and offering solutions that are known from practice. Hence, it is likely that developing these frameworks further would require gathering extensive experimental data.

People have argued that serious improvements may be made by using complementary information channels. For example, displaying graphics along with a dialogue effectively enables more control over a dialogue's context, making it easier to manage. Also, humans use various non-linguistic cues to complement their utterances, and processing these may lead to further improvements (Nagao and Takeuchi, 1994). This leads to the idea of multimodal dialogue systems and virtual environments (VEs). Unlike regular dialogue systems, such a system may involve concurrent dynamics. In fact, the architecture and dynamics of VEs may be highly complex, and various specification and structuring techniques exist to manage this complexity.

The graphical user interface (GUI) is the most basic form of VE which depicts the domain in the form of 'passive' manipulable objects, but the aforementioned complexity issues already show up here. In fact, most of the existing complexity managing techniques were developed for use in GUIs. Like the dialogue system, the GUI was designed for non-computer-experts. It attempts to provide 'natural' interaction by mimicking physical objects. Such a system may be combined with a dialogue system to obtain a multimodal system, in which natural language, pointing, and graphics may be combined.

The VE concept may be taken even further, with a VE containing manipulable objects, and multiple dialogue systems which have a physical presence within the world. In fact, our

current project, the virtual music centre (VMC) does so. It is also available through the WWW, making multiple simultaneous users possible as well. This combination of network VEs with one or more users and dialogue systems is less established, and general frameworks need to be developed.

1.3 A framework for developing natural interaction

The overall goal of my research is to develop a framework for the design of interaction patterns for dialogue systems. I have chosen to emphasise modelling of multimodality and environmental awareness, because, as argued, a lot of progress might be made here. In general, a framework could provide:

- Specification (or description) techniques. A specification language or several such languages may be defined.
- Standard libraries. These include design guidelines, solution templates, and software libraries.
- Evaluation techniques. These include evaluation setups and metrics.
- Methodologies. These are prescriptions of the development process, in other words, which kinds of specification or evaluation to use in what order.

I have started by developing a specification technique for VEs which should provide the basic facilities for structuring VEs and providing environmental awareness to dialogue systems situated within them. This is done by reviewing, trying out, and extending existing specification techniques. The specification technique thus obtained may then be extended with standard shorthands and libraries. These may be obtained from existing frameworks for GUIs and dialogue systems, or from first-hand experience. This is a subject of future research. After that, evaluation techniques and a methodology may be defined. These, however, require HCI expertise as well as more extensive practical experiments. I expect that these will mostly fall outside the scope of my project.

A specification technique may consist of one or more specification languages. Each language may be more or less *abstract*, which means that it explicitly leaves out certain aspects of the system being described. Given a specific problem domain, a specification technique should be:

1. Well-behaved. Intuitively, each specification language that is part of the technique should best be highly unambiguous. Its relation to reality and the other languages should be well-defined. A language that is or can be unambigously defined by means of a formal semantics, coupling it to something well-understood, such as predicate logic, may be called a *formal language*. Arguably, there is no fundamental difference between 'formal specification languages' and programming languages, since expressions in a programming language (should) unambiguously define computational behaviour. Often though, the level of detail of programming languages may be considered too high to be tractable (see the tractability properties below), and it is still useful to have additional, more abstract, specification languages. High-level programming languages, such as functional languages and Prolog, may be seen as lying on the border between 'specification' language and 'programming' language.
2. Expressive. It should be theoretically possible to describe what needs to be described in the domain. Some languages have limited expressivity, making it theoretically impossible to fully describe some systems that can be described using other languages.

Weaknesses in one language may be complemented by another language. The fact that a system may be expressed by means of a language does not necessarily mean that the model thus formed is actually useful. In some cases, models become so cumbersome that they are practically useless. The following tractability properties try to cover these situations.

3. Computationally tractable. One cannot write down a large formal specification without making serious mistakes that are very hard to detect without actual testing or other computational feedback. For example, it is at best extremely costly to write a large computer program without testing a few prototypes or abstract versions of the program first (Green, 1990). Worse, people even have trouble writing small specifications (Gurr, 1994) (Stenning and Gurr, 1996). This means that a specification language that is executable has a great advantage over one that is not. Other kinds of computational feedback include: simulation of user behaviour with the system, automatic verification of internal consistency and other formal properties, using theorem provers, and even just automated syntax and type checking.
4. Cognitively tractable. It should be highly readable to the relevant parties (which may include users as well as designers in some cases), and as many relevant aspects of the problem as possible should be obvious upon reading. Often, it is for this reason that some specification techniques provide several languages: each is meant to emphasise a different aspect that is not well emphasised by another. To further stress the importance of this aspect, it should be noted that software developers often rely heavily on external specifications as memory aids ('display-based reasoning'). This happens throughout the development process, and is done more by expert developers (Davies, 1993). Note that, according to this criterium, not all languages for describing models are quite appropriate for use as specification languages. For example, in pure finite-state models, specifications of larger systems (say, systems with thousands or millions of states) may become very large, making them unreadable. Yet, large finite-state models are still computationally tractable. They may instead be generated from another modelling language which is more appropriate for use as a specification language, such as a process algebra.

Note that these properties are similar to the ones found in (Brun and Beaudouin-Lafon, 1995). In this paper, a similar overview and evaluation is given, though more emphasis is placed on HCI specification techniques, and less on software engineering and AI techniques. Though some aspects of these properties may be treated theoretically, particularly the cognitive tractability property cannot be shown well beforehand. So, any technique will also have to be tested with some reasonably large example.

Though there are a great many techniques for specifying interactive systems, a limited number of general approaches may be identified. The greatest common denominator of many of the techniques is to model the system as a collection of separate modules containing private data and explicit specification of their interaction to the outside world. Listed below are some of the ways in which such models may help the software engineering aspect of VE development, which has proven to be particularly difficult. This list should give a first indication of what kind of models we are looking for. The issues listed are some of the current practical issues in the development of the VMC, as identified after some informal observation.

- A strictly modular structure in which each module has both explicit and limited interaction with other modules helps in making software development in general more cognitively tractable, for the simple reason that each module can be meaningfully examined in isolation. For instance, testing, extending, rewriting, and maintenance are made easier.

- Knowledge modelling. Dialogue systems in VEs may require knowledge of the objects in the VE. Specification of the VE objects as modules may make such knowledge easier to model. In particular, formal but abstract specifications might be made of the existing modules, which may be used directly as knowledge by the dialogue system. As a simple example, a list of the names of VE objects and their location and purpose enables linguistic and non-linguistic references to be made to them. As a more complex example, knowledge of the objects' behaviour and possible behaviour enables the dialogue system to keep track of the goings-on in the VE, to have some idea of what is and is not possible, and what objects may or may not be interacted with in specific ways.
- Concurrent interaction. With multiple agents (including the user(s)) operating in parallel, the structure of the VE becomes highly concurrent. In a modular system, concurrency problems can be managed at the level of the modules: each module is a concurrent process. The privateness of the data within each module enables the most basic concurrent data access problems to be managed. Furthermore, it should be easy for the agents to keep track of changes in the relevant information in their environment. A highly-structured model of module interaction should make it easier to model change notifications.
- Distributedness and multiple users. In distributed systems, services may be (temporarily) unavailable. With multiple users, a user may connect or disconnect at any time. It should be easy to describe what happens in these cases. In a modular system, existence and availability of services or clients (such as the users) may be defined at the level of the modules: a module is either present or absent. In a model in which inter-module interaction is modelled explicitly, it should be easier for one module to identify and hence handle the failure of interaction with other modules.

In addition to these software engineering issues, there should be explicit facilities for human factors issues. The modelling framework should enable communication between the different people typically involved in VE development. The general human factors facilities that a development framework may provide are:

- Specification of the conceptual model (how the user sees the system) and task models (what the user wants to achieve). Proper conceptual and task models should be highly comparable with, if not describable by, the languages provided by the framework. In many dialogue system development frameworks for example, modelling the user's goals in some manner is considered essential for determining the system's reaction to the user's utterances (Bernsen et al., 1998).
- Prototyping. The framework should enable prototypes to be built and analysed easily. Prototyping is an important feedback tool in usability engineering (Nielsen, 1993).
- Evaluation. Evaluation may be done by logging and analysing the relevant events, given proper tools to analyse the logs, or by analysing abstract properties of the system, such as number of operations needed to do a task, visibility of relevant information, etc. In dialogue systems design, it is common practice to collect and analyse user utterance data even before the first prototype is built (Sparck Jones and Galliers, 1996).

Intuitively, a system composed out of modules has at least two meaningful zoom levels. As we shall see, some of the modelling techniques specifically focus on one of these levels.

- Micro level: the perspective from within a single module. Some modelling techniques are meant to model the structure and behaviour of a single module, without being specific on how a system composed of multiple modules should be built.

- Macro level: the perspective on the system as a whole. Some specification languages provide overviews of the system (the most obvious one is a dataflow diagram), and some frameworks offer guidelines for structuring the modules, or standard sets of modules.

1.4 Current state of research

My research has started with a literature study, giving a general overview of the fields of dialogue design and HCI. The results of this study have been published as a technical report (van Schooten, 1999).

The first modelling languages tried are the predicate logic notation Z and the process algebra CSP. CSP turned out to be the most interesting of the two, mostly because Z is not computationally tractable, and cannot describe dynamics easily. An attempt as a reasonably complete coverage of the VMC using CSP models is described in a workshop paper (van Schooten et al., 1999). Some further CSP models of Web sites, including the VMC, may be found in (Donk et al., 1999). CSP has a very high computational tractability and enables intuitive description of several aspects of a system, but it has limited facilities for data modelling, and specification of everything as parallel processes is sometimes cumbersome. These findings are described in more detail in section 2.2.

The next step, described here, is to examine more 'advanced' interactive systems modelling approaches, then decide on the next modelling technique to try out. In the next section, an overview is given of existing approaches. This includes a review of some of the more established techniques using the approach, and some examples.

2 Process communication models

The models are classified into object-oriented models, process algebras, interactor models, and agent systems. Agent systems are further subdivided into intentional logics, production rule systems, agent communication languages, agent platforms, and agent architectures.

The properties of the process communication frameworks that will be reviewed here may be classified using the following set of features:

Feature	Purpose
communication protocols	interface compatibility
update reactions	data dependencies
knowledge base	uniform knowledge model
task negotiation	flexible architecture
lookup service	open systems
routing and migration	network communication

Communication protocols specify which messages processes may send to each other, and in what order. This is the most basic feature, and the main feature that process algebras and object models provide.

Some models provide mechanisms for reacting to updates, that is, propagating the effects of a value change occurring within a process through the system towards the relevant processes. Interactor and dataflow models provide this feature, as well as some agent systems.

A 'knowledge base' (KB) is often mentioned as an important feature of an agent in multi-agent systems. A KB typically stores a number of facts, and often it also contains rule sets which define the behaviour of the agent. The KBs of the different agents are typically specified in the same language. Each agent's behaviour may thus be specified in a uniform way, and the agents may exchange knowledge in a uniform way.

Some agent models provide task negotiation. This means they have facilities for modelling tasks and activities, possibly including plans, alternatives, subtasks, task quality, task failure, etc., incorporating the planning mechanisms known from classical (single-agent) AI. Agents may request others to do tasks for them, request their capabilities, negotiate to decide on a multi-agent plan, etc. This way, a task may be achieved in a more flexible way, accounting for the agents that happen to be present in the system, and possible failure. Some models also model concurrent, possibly conflicting, tasks. Note that existing task negotiation models are still limited. Often, negotiation strategies consider only task quality and task failure handling, rather than task structure.

An agent lookup service is usually a service provided by a separate agent, which maintains knowledge about the other agents in the system. When given a description of properties, it looks in its database for agents matching these properties. This feature is useful in systems with constantly varying numbers of agents, i.e., 'open' systems.

Migration is the relocation of a program from one computer to another, without disturbing its execution. Routing is the passing of messages through computer networks. With routing and migration, the physical medium of the agents (the computer network) becomes visible in the model, so that practical network communication problems (especially efficiency) may be addressed.

2.1 The object-oriented model

Many of the concepts and mechanisms found in process algebras and agent systems are also found in object-oriented (OO) systems. When viewed as a model of communicating processes, the OO model (Booch, 1994) is one of the oldest such models, dating back to 1967 (Dahl et al., 1968). It may also be considered the most well-established one. The model that is usually seen as the one that made OO development popular is Smalltalk-80 (Goldberg and Robson, 1989).

The conceptual view of OO as a model of communicating processes, rather than of abstract data types, is clearly found in the examples in the Smalltalk book cited above. This point of view is actually very close to that of the more recent 'agent-based' modelling techniques, described in section 2.4. In fact, the examples in the Smalltalk book look much like typical examples of agent-based modelling (see for instance (Jonker and Treur, 1998c)).

An object consists of a number of internal variables, determining the state of the object, and a number of methods, which have specific names. A method depicts a specific procedure, which may have parameters and a return value, and which operates on the internal variables. Different objects may have different methods with the same name. Each object also has a class. If two objects have the same class, they also have the same methods.

Any object which has a reference to an object may call methods in it by calling the methods by name. Calling methods is also called message passing. The subset of methods that are actually available through a reference may be determined by the class of the object it refers to, the class of the reference, and the class of the object which holds the reference.

The classes are generally ordered by some classification scheme: a class may be defined by inheriting properties from other classes, that is, by importing method names, methods, and variables from them. If a class is defined in terms of other classes, we will call it the subclass of those other classes, and we will call the other classes the superclasses of the class. It is possible for a class to define method names without defining the methods themselves. Such a class is called abstract. The actual methods may be filled in by subclasses of the class. If the model does not provide a classification scheme, it is called object-based rather than object-oriented.

Each non-abstract class has exactly one corresponding class object. The class object is a factory of the objects of the given class: it always has at least one method which initialises an object of the class it stands for, and returns a reference to it. This way, an arbitrary number of objects of the class may be created through calling this method. Class objects may also have extra methods and variables for managing the objects of the class. Typically, all objects always hold references to all class objects.

At any point during the execution of a program, some method is being executed by the computer. We will call this 'locus' of execution a thread. It is possible that several threads are running simultaneously on different computers, or on the same computer. If no thread is running inside an object, and there exists no reference to the object by any object which does have a running thread, we can be sure that the object cannot be used again, and it may be deleted. The object generally cannot delete itself, nor can it force references to it to be dropped.

Though the idea of method calling as 'message passing' suggests the ability of an object to execute concurrently, there is no separate thread for each object. Instead, each object is reactive, reacting immediately to a message passed to it, and making the sender wait until it is finished. This uniquely defines a control flow that can be modelled by a single thread. In the case of multiple threads, a thread is not coupled to a specific object, but is itself an object of a separate *thread* class. So, perhaps somewhat counterintuitively, multiple threads

may be running inside a single object.

2.1.1 Discussion

Whether the OO modelling technique is actually a good specification technique in general remains an open question. Comparisons of software built using OO with software built using more traditional programming techniques only yield inconclusive results (Smith et al., 1995). Some causes of problems have been pointed out in the literature:

- OO design invites a fine-grained modular composition, which means that the different modules will be closely coupled. Experiments show that trying to understand the behaviour of a typical object class is a difficult process (Kung et al., 1994). This is apparently because it requires understanding of the many other classes that it uses, and the ways in which it is used by other classes. Furthermore, the control flow of the program as a whole is hard to follow, since the locus of control is constantly jumping between objects (Agarwal et al., 1996).
- When using inheritance, the behaviour of a class depends on its superclasses. This means that understanding an inherited class requires understanding of its superclasses (Daly et al., 1995). Furthermore, the optimal inheritance structure crucially depends on the nature of the problem domain. If the domain is relatively unknown, inappropriate choices may be made easily (Riel, 1996).

Both the close coupling between modules and inheritance result in inflexible and hence unmaintainable programs (Kung et al., 1994).

As a final remark, it should be noted that the original Smalltalk language is closely integrated with a GUI environment, in fact providing one of the first GUI description models. All GUI components (buttons, windows, etc.) correspond to objects, which send messages as a result of the user's operations on the components. Nowadays, many GUI programming libraries are arranged in a similar manner. Possibly, one of the more important contributions of the OO model is the model it has provided for specifying GUIs.

2.2 Process algebras

An informal description of the process algebras CSP and CCS will be given here. In particular, they will be conceptually compared to object-oriented and other procedural programming languages. This will lead to process algebraic models of the object-based communication scheme, described in section 2.2.2.

Process algebras describe systems as a number of parallel processes, communicating through channels which are identified by names. A process is like an object, but always has exactly one thread. Like an object, a process may be created by another process. Unlike the OO model, this automatically creates a new thread for this process. Furthermore, a process is able to terminate, effectively deleting itself.

The dynamics of each process are described explicitly, in a form similar to that of a finite state automaton. As long as the total number of states of the whole system remains finite, the system's dynamical behaviour can be verified exhaustively: this is one of the main advantages of process algebras.

Communication between processes proceeds through channels, which are identified by names. To initiate participation in communication, each process signals that it is ready for a specific

set of channels, called its nextset. The possible communications can be determined only when the nextsets of all processes in the system are known.

In CSP (Communicating Sequential Processes) (Hoare, 1985), communication is multi-way and symmetrical. In order to determine how many processes a channel is supposed to synchronise with, the set of channels that each process might participate in during its execution must be known when it is started. This set is called its alphabet, which may not change during the lifetime of the process. A process' nextset must always be a subset of its alphabet. Only when all processes that have a channel in their alphabet also have it in their nextset, communication over this channel may proceed.

In CCS (Calculus of Communicating Systems) (Milner, 1980), communication is two-way and asymmetrical. Each process is either a reader or a writer of a channel, and communication may proceed as soon as a channel has one reader and one writer. Therefore, it is not necessary for the processes' alphabets to be known in order to determine the system's behaviour.

In both CSP and CCS, communication is modelled as a synchronous, atomic action, in which all participants participate simultaneously. All participating processes will then have to submit new nextsets. All non-participating processes will remain blocked. In case communication over several channels is possible, one channel is typically chosen randomly.

In CSP, data passing is not conceptually different from channel synchronisation. Data passed through a channel is modelled as an array of channels, obtained by concatenating the original channel's name to the representation of each value of the data's domain (for example, passing a byte could be represented by 256 channels, ending with the strings '000'...'255'). Writing a data value through a channel can then be modelled by requesting one specific channel out of the array, and reading though requesting any channel out of the array. A reader could specify only a subset of the values, thus forbidding any other values to be written. This last possibility may be seen as something 'in-between' reading and writing, since there is no fundamental difference between the two. The value or value range specified by each process may be instead seen as a *limiting constraint* on the value that is eventually chosen.

CCS has slightly different data modelling capabilities. Writers always write specific values. Readers may simply read any value offered, or they may specify a constraint on the offered value. If the constraint does not hold, the reader will refuse to read. Unlike CSP, another reader is still allowed to read the value instead. The value or value range specified by each reader may be seen as an *enabling constraint* on the value that is eventually written.

In both CSP and CCS, it is possible to create new processes recursively. In order to be really useful, this should be combined with some scheme to create new channels along with the processes. One scheme that is found in the CSP language are the channel hiding and renaming constructs. When a process is started, a set of channel names may be specified that may from then on be used for private communication between the started process and its creator. This is called channel hiding. For convenience, any process may also have some of its channels renamed at startup. This scheme is limited, since the communication structure of the system may not be changed arbitrarily. See section 2.2.2 for alternative models.

2.2.1 Discussion

As mentioned in section 1.4, an evaluation was done of CSP as a specification language. The dialect used was a minimal subset of CSP, that is, without channel hiding or shorthands for value passing. Basically, this means that a system is specified as a fixed number of parallel state automata. For the purposes of this general evaluation, this was considered sufficient. Some conclusions are given here. Advantages are:

- System modelling. Both the system architecture and the module interfaces are specificed explicitly.
- Task modelling. CSP may be used to specify task hierarchies in a natural way, and is nearly as expressive as more established task modelling techniques, such as GOMS. Furthermore, the task model may be executed with the system model.
- Generation of prototypes. A CSP description may be coupled to an implementation language, by making some CSP events trigger function calls, and translating some user input events to CSP events. This way, a prototype may be generated which illustrates both dynamical and appearance aspects.
- Verification. Many system dynamics properties may be verified using model checking. This may be used for both tracking bugs and, possibly, formally identifying usability problems.

Limitations are:

- The requirement to write everything as parallel state automata is sometimes cumbersome. In particular, two problems may be identified:
  - A CSP specification of the behaviour of a complex agent as a state automaton may become very large or has to remain abstract. The CSP way to make such a specification smaller is by splitting the agent process into multiple processes. However, it is not clear how, or whether, this should be done. In other words, one is forced by the language to specify complex agents in a specific, not necessarily intuitive, way.
  - Having to ensure that the processes have the correct alphabets with each incremental change during the specification process turned out to be a nuisance. When adding or temporarily commenting out particular processes, changes in other processes' alphabets sometimes needed to be made. A significant part of the errors made during specification were alphabet problems.
- Limited facilities for data modelling. It is not practical to model the passing of complex data with each event, nor is it practical to model processes that maintain a lot of data internally. A complementary notation including both data and dynamics as two separate aspects of the specification, such as found in LOTOS (Bolognesi and Brinksma, 1987) or (Fischer, 1997) might have been used. However, in both these techniques, the facilities for verification of the data aspect are limited.
- The ability to model processes with arbitrary point-to-point communication channels is important, if not essential, for specifying systems with dynamical numbers of processes which have direct communication other than those between parents and children. This is cumbersome to specify without extra shorthands. Ways of modelling such communication are addressed in section 2.2.2.
- The central synchronous communication model is inefficient in distributed environments. This means that the dynamics of the system may not be directly transcribed to the final system, but will have to be modified or automatically transformed. A simple solution to this would be to prohibit multi-way communication. The synchronisation model thus obtained is a 'greatest common denominator' between those found in CSP and CCS.

In retrospect, CCS might have been an improvement over CSP in terms of cognitive tractability. The original reason why CSP was chosen instead of CCS is that CSP descriptions map

more closely to logical constraints. This makes it easier to describe additional (limiting) external constraints on a given system by just adding extra processes, and makes it easier to transcribe the system to logical specifications.

However, it should be noted that the concept of multiple synchronisation constraints or any form of multi-way synchronisation at all was *not* used in the specifications that were actually written. This also means that the alphabet specification problems just mentioned were more of an inessential nuisance than a part of the fundamental problems of the systems being specified. It may still be argued that the separate alphabet specification may be useful as a kind of double-checking, similar to type checking, enabing the computer to point out some programming errors clearly and effectively. However, unlike type checking, a process that specifies the wrong alphabet results in different system behaviour, rather than a clear deadlock situation or other kind of clear error indication. Furthermore, the only kind of double checking that may actually produce compiler errors is a mismatch between the process's alphabet specification (if given separately), and the actions used in the process's behaviour specification. Type checking is typically a cross-check between a caller's call format and a callee's call format, reducing the need to keep browsing back and forth between the caller's and the callee's specifications. Alphabet checking only double-checks two specifications belonging to the same process, which is less useful.

Another problem was found while writing the CSP specifications: in several places, it was necessary to work around the CSP system in order to model 'one reader multiple writers' scenarios. These are very easy to do in CCS, as they fit naturally into its asymmetrical synchronisation model.

As a final and very general note, the CCS model is closer to traditional procedural programming languages than CSP. This means that transfer of programming knowledge may be easier for CCS for many people. In CSP, adding a process means limiting the behaviour of the rest of the system. In terms of procedural programming, this is very strange: when a process is *removed*, the rest of the program does *more* actions.

2.2.2 The object-based model in process algebra

To clarify the correspondence between object models and process algebras, the object-based model will be described in CSP and CCS here. The object-based model offers a way to describe arbitrary point-to-point communication, addressing the problem mentioned in section 2.2.1. The models described here may serve as a first step towards a more formal account of the dynamics of more 'advanced' models.

It is not directly possible to model private point-to-point communication between two processes, other than between a process and its creator. To achieve this, one would have to add an explicit notion of 'pointers' or 'references' to the model: it must be possible to pass some kind of reference to a private channel as data along another channel. The process receiving the reference may then access the channel through the reference. Such a scheme may actually be seen as a complete substitute for channel hiding.

Such a scheme is found in π-calculus (Milner, 1993), which is an extension of CCS. A more simplistic but readily model-checkable scheme, which is closer to the object-oriented scheme, is described here. In this scheme, identities of objects are passed, rather than those of channels.

The pointer scheme we will adopt here is the scheme found in the object-based model. This means each process is given a separate set of channels, and a unique identity, such as a number. The process' channels may be used as soon as another process obtains the process' identity. Separate sets of channels might for example be modelled using some concatenation

scheme, similar to that used for data: each process then obtains a separate set of channels by concatenating its identity to the channel names it uses. Communication through a channel, whose name is obtained by concatenating the method name to the object's identity, is analogous to a method call. In each method call, one value may be read or written.

In CSP, enabling the possibility of point-to-point communication between any two processes implies that there has to be a separate channel for each pair of processes that wish to communicate privately. When using a concatenation scheme, this means that not only the referred-to process, but also the referring process should be concatenated to the channel's original name. Each process should then have its alphabet determined by its identity: it should be able to refer to any channel with itself as referrer, and be referred to through any channel with itself as referred-to process. Given that all channels of a given CSP system (without channel hiding) are defined as concatenations

<base-name> <referrer> <referred-to> <data>

with

(<base-name> , <data>) $\in \mathit{ChannelSpec}$
<referrer> $\in \mathit{ProcID}$
<referred-to> $\in \mathit{ProcID}$

with *ChannelSpec* being the complete set of methods within the system, a process with identity ID $\in \mathit{ProcID}$ should have as its alphabet

<other-base-name> ID <referred-to> <other-data> $\cup$
<own-base-name> <referrer> ID <own-data>

for any

(<other-base-name> , <other-data>) $\in \mathit{Accessible} \subseteq \mathit{ChannelSpec}$
(<own-base-name> , <own-data>) $\in \mathit{Alphabet} \subseteq \mathit{ChannelSpec}$
<referrer> $\in \mathit{ProcID}$
<referred-to> $\in \mathit{ProcID}$

with *Alphabet* being the process' own set of methods, and *Accessible* the set of others' methods that the process has access to. Each process should be spawned by a 'class factory' process, which has as its alphabet all channels that all processes of its class will ever use.

But, we are not finished. Each process, when started up, gets the potential to call or be called by any other process, whether this other process is already in existence or not. As long as some of the callers do not yet exist, they will not block the process from executing its own methods privately. A solution to this is to make the class factories block all channels belonging to callers that do not yet exist.

Specifying such a model efficiently depends on the facilities and shorthands offered by the model checker used. Since the model checkers examined up to now are not powerful enough to deal with this kind of composite channels, the above model has remained sketchy. Instead, we will look at the CCS version first, which turns out to be more readily implementable.

In CCS, adding arbitrary point-to-point communication is easier. We need not worry about alphabets, and all that is required is one set of channels per process. This is enough to enable each method call to be a one-reader-multiple-writers, or a multiple-readers-one-writer situation, where the 'one' always stands for the process that is called. In the CCS-based verification language Promela (Holzmann, 1991), it can be easily described using the channel array construct. In Promela, an unbuffered channel is defined by: chan <channame> = [0] of <datatype>. An array of channels may be defined by chan <channame> [<nr_of_chans>] = [0] of <datatype>. Specifying the readiness to write to a channel is specified by <channame> ! <value>, while readiness to read is specified by <channame> ? <variable>. A 'while' type loop is defined by do :: <statement1> ... :: <statementN> od. This

defines N statements, one of which is chosen to be executed for each iteration of the loop. The `;` and `->` operators signify sequential execution of actions. The program always starts by executing the `init ...` process. Other constructs are much like those found in the programming language C.

```
/*** The methods, defined as arrays of channels ***/

  #define MAXPROC 15    /*** maximum no. of processes in the system ***/

  chan method1[MAXPROC]        = [0] of {datatype1};
  chan method1_return[MAXPROC] = [0] of {returntype1};
  ...
  chan methodN[MAXPROC]        = [0] of {datatypeN};
  chan methodN_return[MAXPROC] = [0] of {returntypeN};

/*** The ID handler and its channels. Each process and its creator obtains
 *** its ID through communicating with the ID handler. ***/

  chan requestid = [0] of {byte};
  chan returnid = [0] of {byte};

  proctype id_handler() {
    byte nextid;
    nextid=0;
    /*** An endless loop, in which the handler waits for a process' creator
     *** to query the ID of the process just created (returnid) and then
     *** expects the new process to read its ID through requestid. ***/
    do
    :: returnid!nextid -> requestid!nextid -> nextid++;
    od
  }

/*** The processes ***/

  proctype p1(parameters) {
    byte my_id;       /*** the process' identity ***/
    ...               /*** other internal data ***/

    requestid?my_id;  /*** Process reads its ID ***/
    ...               /*** other initialisations ***/

    /*** The process' set of methods. In order to comply to object-based
     *** dynamics completely, each method call has a return synchronisation,
     *** so that the caller will remain blocked until the callee is finished.
     *** Unlike object-based models, we might have specified different
     *** or more restrictive temporal ordering constraints. ***/
    do
    :: methodI[my_id]?data -> ... -> methodI_return[my_id]!return;
    :: ...
    :: methodJ[my_id]?data -> ... -> methodJ_return[my_id]!return;
```

```
    od
  }

  proctype p2 (...)

  ...

/*** The startup procedure ***/

  init {
    byte first_id;    /*** reference to the first process created ***/

    run id_handler(); /*** start up the ID handler ***/

    /*** When creating a new process, a new ID must first be requested
     *** through returnid. Then, the process is created, and reads
     *** its ID though requestid. ***/
    returnid?first_id;
    run p1(...);
    ...

    /*** This is how a caller should call a method ***/
    methodK[first_id]!data -> methodK_return[first_id]?return;
    ...
  }
```

The state space remains finite, and the model can be checked exhaustively. The state space is also smaller than in the CSP model, because, for a maximum of n processes, only n sets of channels are needed, rather than n^2. Note that process deletion is not accounted for: the ID handler will eventually run out of numbers, even if the total number of processes remains limited. This is not hard to add: the ID handler will need to keep track of the existing processes, for example using a private boolean array, and each process should signal its deletion before it exits. Note that real garbage collection, in which processes are automatically deleted, would be harder to add.

Note that some semantic constraints present in object-oriented models are not enforced. In particular, a process is not prohibited to use a channel without obtaining a valid reference first. This reference model is closer to the 'C++' reference model rather than a real object-oriented one. This constraint could be modelled by specifying a new 'reference' data type which imposes limitations on assignment of variables of this type.

It is possible to model OO style concurrency, with threads as separate processes. A new thread may be created by creating a process which has a method that keeps running after its return synchronisation. After creation, the thread is started by calling the method. The other processes should all retain their regular call-return structure. This is similar to Java's threading scheme. Process algebra style concurrency may be introduced simply by having the processes deviate from the strict call-return synchronisation.

2.3 Interactors and dataflows

While object models and process algebras form a basis for describing interactive systems, interactor and dataflow models have extra features for making change propagation through

an interactive system easier to manage.

Interactors are basically a special kind of objects. Examples of interactor models are Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC) (Hussey and Carrington, 1996). In both models, there are interactors for storing actual data (the Model and the Abstraction interactors) and objects for viewing (View and Presentation) and modifying (Controller and Control) the data. Changes made to data objects through the modify interactors propagate through the object structure in restricted ways. The system is thus composed out of 'packages' of locally-modifiable data, with notifications of changes to this data flowing through the system to the relevant places, according to some well-defined propagation scheme. See fig. 1 for an example.

The main difference with respect to regular object-oriented models is that interactors provide standard facilities for defining a call structure that is different from the more usual object call structure. For example, they may provide callback schemes, which enables an object to call a method in its creator, rather than just the other way round, or registration schemes, in which an interested party can register for specific update messages. Interactors also specify restrictions upon this structure, so that the system dynamics become more manageable. For example, PAC prescribes a strictly hierarchical structure. The more established interactor models also provide standard libraries, in the form of software libraries.

The VR specification language VRML (Carey and Bell, 1997) uses a 'dataflow' model that is conceptually close to interactor models. The VRML world consists of nodes, which contain data. Some nodes have specific input and output points. An event arriving at an input point triggers a procedure inside the node, which may change the node's data, and produce events through one or more of the output points. The most basic case is the simple data node: it has one input point for changing its value, and one output point for notifying its value changes to other nodes. There are also output points that spontaneously generate events, such as timer and user-triggered events. Arbitrary routes connecting input points to output points may be specified separately, by means of `ROUTE` commands, as long as each input's data type matches each output's. Each input/output point may have several routes leading to/from it. A similar 'routing' model is found in Java, using events and event listeners. The `AddListener` method is analogous to the VRML `ROUTE` command.

Sensors may be declared for sensing user click and drag operations on groups of nodes. In VRML, all nodes (including physical objects) are conceptually arranged in a hierarchy. Each sensor enables and detects operations done on its sibling nodes, and on any nodes further down the hierarchy. A sensor's sphere of influence may be overridden by another sensor situated further down the hierarchy, see fig. 2. This model is analogous to the X11 event propagation model, see fig. 3.

An advanced dataflow model is found in the GUI development kit Amulet (Myers et al., 1997). Here, each GUI object has a number of attributes with specific names. Attributes may be added or deleted at runtime. The names of the attributes determine their meaning. Each attribute has a value, or a formula that determines its value. The formula may be specified in terms of other objects' attributes. Stated in VRML terminology, the formula implicitly defines routes from the other objects' attributes to a script, which re-calculates the attribute each time some value update arrives.

Like VRML, Amulet provides a grouping construct, which may be used to define a system as a hierarchy of objects. Unlike the other models, Amulet's formulas may be explicitly specified in terms of the hierarchy. The following example (adapted from the Amulet manual) illustrates this:

```
Am_Define_Formula(int, owner_width)  {return self.Get_Owner().Get(Am_WIDTH);}
```

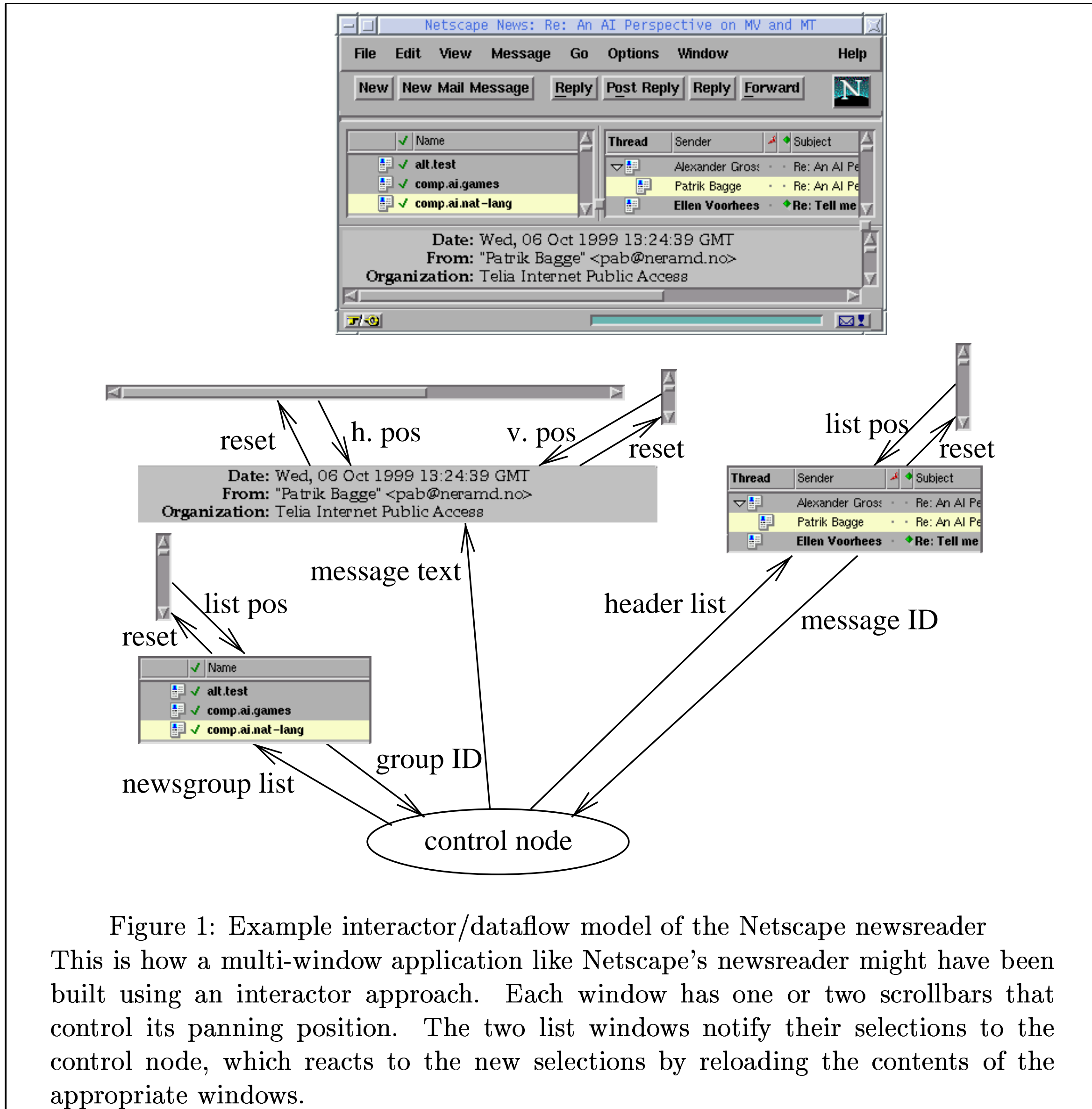

Figure 1: Example interactor/dataflow model of the Netscape newsreader
This is how a multi-window application like Netscape's newsreader might have been built using an interactor approach. Each window has one or two scrollbars that control its panning position. The two list windows notify their selections to the control node, which reacts to the new selections by reloading the contents of the appropriate windows.

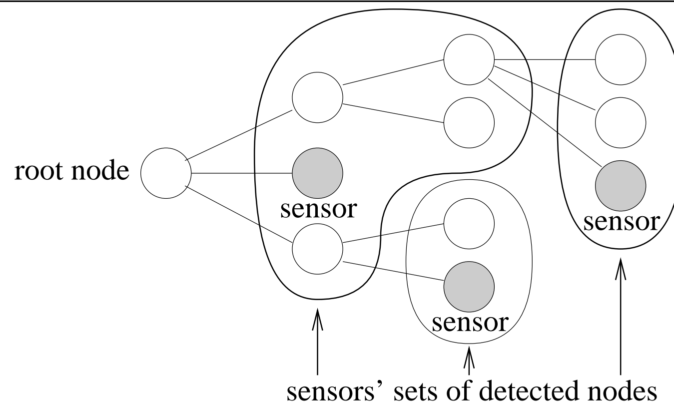

Figure 2: VRML's sensor detection scheme
Each node specifies a physical object or a 'special' node, such as a script or sensor node. The nodes are arranged in a tree. A sensor node (grey) enables clicking or dragging of all physical objects within the same subtree as the sensor node, including all objects further up the tree. Sensor nodes are overridden by sensor nodes located further up the tree.

```
Am_Define_Formula(int, owner_height){return self.Get_Owner().Get(Am_HEIGHT);}

Am_Define_Formula (int, arc_width)
    {  return self.Get_Owner().Get(ARC_PART).Get(Am_WIDTH);  }
Am_Define_Formula (int, arc_height)
    {  return self.Get_Owner().Get(ARC_PART).Get(Am_HEIGHT);  }

Am_Object my_group = Am_Group.Create ("my_group")
    .Set (Am_LEFT, 20)
    .Set (Am_TOP, 20)
    .Set (Am_WIDTH, 100)
    .Set (Am_HEIGHT, 100)
    .Add_Part(ARC_PART, Am_Arc.Create ("my_circle")
              .Set (Am_WIDTH, owner_width)
              .Set (Am_HEIGHT,owner_height))
    .Add_Part(RECT_PART, Am_Rectangle.Create ("my_rect")
              .Set (Am_WIDTH, arc_width)
              .Set (Am_HEIGHT, arc_height)
              .Set (Am_FILL_STYLE, Am_No_Style));

my_win.Add_Part(my_group);
```

The first four statements define some formulas returning integers. The first two return the width and height of the object's owner (=parent), the second two return the width and height of the `ARC_PART` attribute of the object's owner.

The rest of the statements define a group object with a circle and a rectangle inside it. The width and height of these are defined using the previously defined formulas: the circle gets its size from the group object, and the rectangle gets its size from the circle. If the group object's size were changed, the circle and rectangle would automatically adapt their sizes. If, say, the rectangle were placed inside another group object, it would try to find an arc_part

inside its new owner, and adapt its size to this new object's size.

Such structure queries may be used to design components with some degree of flexibility: for example, Amulet defines standard interactor components that either operate on any of their sibling objects (analogous to the VRML sensors), or, if they can't find any siblings, they operate on their owner instead.

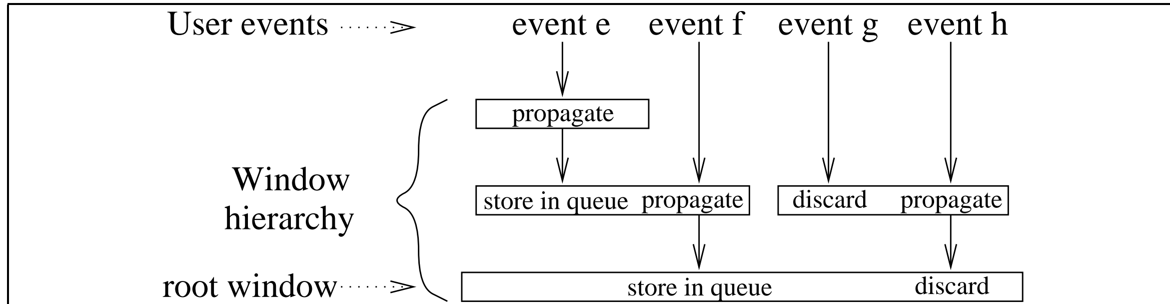

Figure 3: X11 event propagation mechanism
X11 has a hierarchical window structure. Each window except the root window is contained within one other window, called its parent. The user generates events, which are located at specific windows. Each window is subscribed to specific types of events. If an event is generated that a window has subscribed to, it is stored in the window's event queue, to be processed later by the process that manages the window. If a window has not subscribed to the generated event, it is propagated to its parent. Windows may also choose to discard events and are able to spontaneously generate events.

A model specially made for systems with nodes operating in parallel is Reactive-C (Boussinot et al., 1998), and its more recent Java implementation, SugarCubes. In this system, a node may be connected to one 'sync' node, denoting a synchronous area. All nodes connected to the sync node execute in synchrony, using a discrete-time model. Events may be sent to the sync node, which will multi-cast the event synchronously to all other connected nodes. Each of these nodes may have chosen to subscribe to specific types of event only, and will discard any other types of event. A node may also send an asynchronous message to a sync node. In this case, the delay between sending and delivery may be arbitrary. See figure 4.

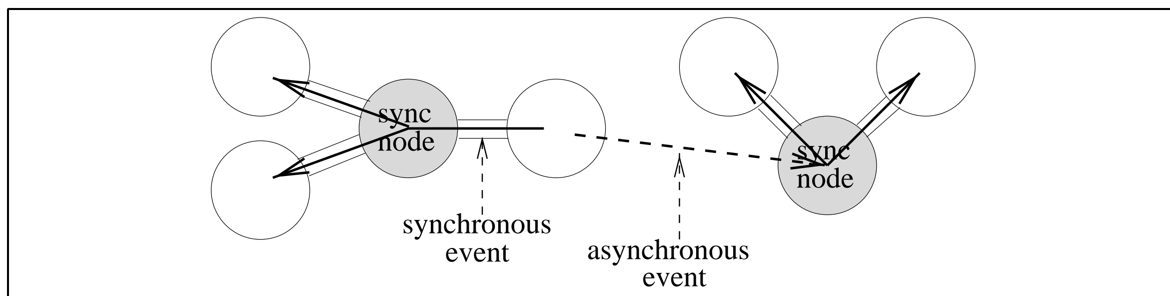

Figure 4: Reactive-C event model
The figure shows two sync nodes with nodes attached, indicating two synchronous areas. Synchronous messages (thick line) may sent to a sync node, which sends it to the other nodes within the same time instant. Asynchronous messages (dotted thick line) may be sent to other areas, but may have an arbitrary delay.

2.3.1 Discussion

We may distinguish two kinds of events: explicit and implicit events. In some models, nodes may explicitly send events. An event may stand for anything: a message, a mouse click, a creation of a new object, or a change of a variable. The reaction of receiving nodes may also be arbitrary. In some models, in particular Amulet, the effect of some events may be specified implicitly, leading to a more concise notation. In Amulet, value dependencies may be defined in the form $DependentValue = f(value_1, \ldots, value_n)$. Any changes in any of the values $value_i$ implicitly generate events to update $DependentValue$. Further events will be generated as a result of the change in $DependentValue$. This does mean that the dependency structure should not contain cycles, otherwise the system may diverge. Preferably, the path from any node to any other node should also be unique. If it isn't, race conditions may occur because a node may receive multiple updates as a result of a single value change. See fig. 5.

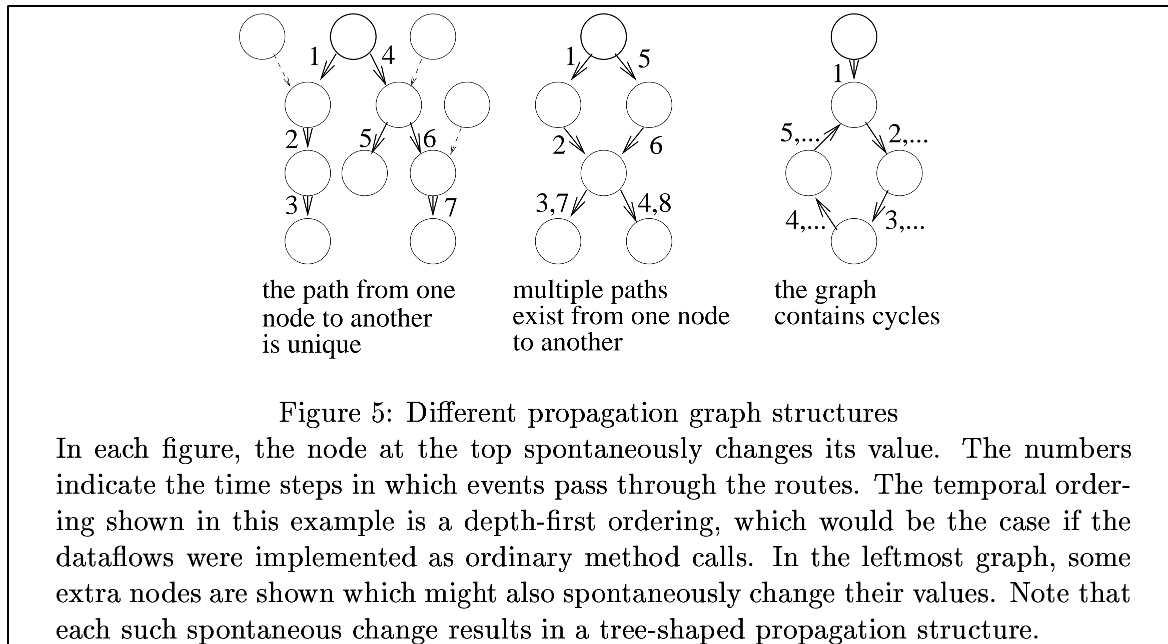

Figure 5: Different propagation graph structures
In each figure, the node at the top spontaneously changes its value. The numbers indicate the time steps in which events pass through the routes. The temporal ordering shown in this example is a depth-first ordering, which would be the case if the dataflows were implemented as ordinary method calls. In the leftmost graph, some extra nodes are shown which might also spontaneously change their values. Note that each such spontaneous change results in a tree-shaped propagation structure.

We may distinguish two kinds of routes: explicit routes and implicit routes. In addition to the usual explicit routes, some of the models provide a hierarchical structuring mechanism. This structure implicitly determines the existence of additional routes. In X11 and VRML, we find the following ones:

- Positioning. In both systems, the positions of children are typically relative to the position of the parent.
- Value sensing and event propagation. Event sensors may be placed somewhere in the hierarchy, which will sense anything within their sub-hierarchy.

These structuring facilities are mostly hard-coded. Only Amulet provides a general model for implicit routes: it provides facilities for nodes to query the structure in their dependency specifications. This leads to a more 'declarative' specification, declaring routes in terms of properties of the node structure, rather than laying all routes directly to specific nodes.

Many interactor models do not give a detailed account of their dynamics. To some extent, the constraints that the models place upon system structure relieves the need for a precise dynamics description. The dynamics may be classified into (roughly) five types (see fig. 6):

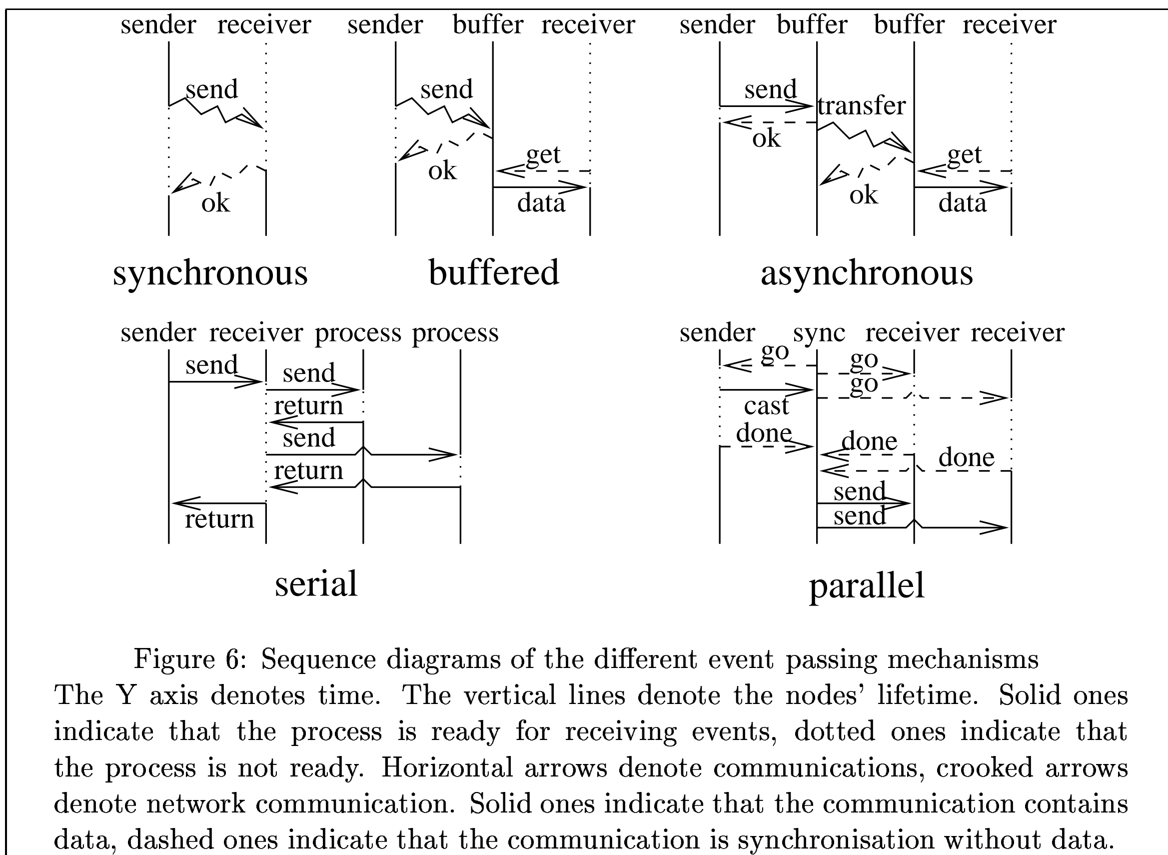

Figure 6: Sequence diagrams of the different event passing mechanisms
The Y axis denotes time. The vertical lines denote the nodes' lifetime. Solid ones indicate that the process is ready for receiving events, dotted ones indicate that the process is not ready. Horizontal arrows denote communications, crooked arrows denote network communication. Solid ones indicate that the communication contains data, dashed ones indicate that the communication is synchronisation without data.

- **Synchronous**. In a synchronous model, each event must be handled as it arrives, and the sender has to wait until the event is actually received by the receiver. This guarantees that all events arrive in the order they were sent. The synchronous model is the basic model that process algebras provide.

  In the most basic case, each node is reactive, i.e. it handles the event and gets ready for receiving the next event or sending the appropriate output events immediately. This means that a receiver is not always ready for receiving all types of event, and senders may have to wait until it is ready.

- **Buffered**. A special case of synchronous communication is when communication is buffered. This means that incoming events are stored in a queue which is part of the receiving node, which are handled by the node eventually. This means a sender never has to wait for a receiver to reach a ready state. Process algebras typically provide standard shorthands for describing buffered communication.

- **Asynchronous**. Asynchronous models are useful for distributed communication. In cases where it is a requirement that a sender never has to wait for network delays, it will typically store its outgoing events in a local event queue, which is transferred eventually. This means that events may take arbitrary time to arrive, and events from different sources may arrive out of order. This is the model found in X11's client-server communication model.

  Reactive-C provides an even less constrained asynchronous model. This model specifies that even events from the same source may arrive out of order. This happens in practice when part of the data during a network transfer is lost and re-sent, re-shuffling the order of arrival of the data.

- **Serial**. Events are handled one at a time, and each is propagated entirely before the

next one arrives. This happens in most models, including most interactor models, Java's event listener model, (approximately) VRML, and Amulet. Method calls with return synchronisations, as were described in the Promela model in section 2.2.2, are an instance of the serial model. After each send action the sender is forced to wait for a return synchronisation, which signals that the event is handled completely by all relevant processes. If it is ensured that each event generates only a tree-shaped call structure, no concurrency problems exist, because execution order is not important, and serialness prevents any new event to arrive before the current one is handled completely.

- **Parallel**. Events may be generated simultaneously, and may be handled simultaneously. Reactive-C provides this model. Such a model enables new possibilities. For example, Reactive-C provides a model for resolving cyclic dependencies: within each time step, only one update is handled for each value; subsequent updates are ignored. Furthermore, it is now possible to ensure simultaneous handling of data. For example, in animation, multiple objects often have to stay in exactly the same relative positions when they are moving together.

Unlike object and process algebra models, interactor models emphasise and facilitate macro issues rather than micro issues. They provide:

- Structural constraints, such as the MVC and PAC models, providing structural design guidelines.
- Routes which appear as an independent part of the specification, declaring separately which object communicates to which, making the data dependencies more explicit, and facilitating re-wiring of the dataflow of part of the system from outside.
- Standard facilities that treat the program as a whole as a data structure, in particular the implicit route schemes.

2.4 Agents

Without further clarification, 'agent' would be a confusing term to use. Various different definitions and classifications of agent systems exist (Jonker and Treur, 1998b) (Veer et al., 1998, the lecture by P. Braspenning). The classifications typically define agents in terms of metaphors, such as intelligent, intentional, autonomous, pro-active, learning, cooperating, and social. Some classifications also include the general application areas in which software with such properties may be used, such as information agents, internet agents, and interface agents.

Some argue that the most important aspect of the agent concept *is* its metaphor as a design metaphor (Wooldridge, 1999). The view of a module as 'anthropomorphic', having the possibility of autonomy in its actions and flexibility in its request handling may offer a model for robust software that is easy to understand intuitively. It also fits neatly onto the new possibilities offered by the internet (such as internet search and electronic commerce) and the perceived wishes and needs of the increasing group of computer users who are not computer experts (anthropomorphic user interfaces and 'smart' autonomous applications).

Another advantage of this metaphor that is mentioned in the literature (Veer et al., 1998, the lecture by H. Weigand) is that the idea of complex communication protocols and a database in each module makes it possible to design systems with a coarser compositional granularity than OO systems, since the granularity of OO systems has sometimes proved to be too fine (see section 2.1.1). The practice of wrapping a legacy system inside an agent in the hope of increasing maintainability, called 'agentification', is an example of this idea.

Discussion. One might argue that such an 'agent' paradigm creates new problems:

- It does not seem meaningful to describe a module in terms of complex and possibly vague human metaphors, when the module's behaviour itself is actually clear and simple. For example, the paper (Brazier et al., 1998) addresses an electricity load balancing problem using an agent-based approach. However, one of the most complex issues of this problem is the system's collective convergence to a solution. This issue is actually addressed using control systems theory. Arguably, the problem was not made much easier by the modelling approach used insofar as it was agent-based: plain control systems theory may have been enough.
- All complexity moves towards the communication protocols and data models, which may be very complex and may still become unmaintainable.

  Some examples of agent communication use Prolog as a communication language. One problem that comes to mind is the complexity of the language interpreter needed. Which Prolog version and engine is being used? What if the Prolog interpreter has to be upgraded, for example, because of fatal bugs?

  Another problem is that processing messages in higher-level languages approaches directly executing code that is supplied by another program. The behaviour of a system consisting of modules that communicate by sending arbitrary code to be executed might become very hard to understand.
- The internals of an agent, including any 'flexibility', 'autonomy', or 'intelligence', still have to be programmed.

Nevertheless, we will freely use the word 'agent' for any type of software module or process with private data and an explicit interface, as long as the metaphor fits well. It is not in the spirit of this view to give an overly long definition of 'agent'. In fact, the attempts at definitions found in the literature has been more than sufficient. Further refinement of the concept is perhaps best done by looking at the interactive systems development frameworks that have been proposed.

Development frameworks. In the rest of this section, more description techniques and standard libraries will be described. They are not fundamentally different from those described up to now, but they will particularly include techniques that have emerged or become more in focus because of the recently increased interest in the agent concept. In particular, we will look at:

- Intentional logics. There is a variety of literature that discusses logics that are supposed to be used to specify agents. The distinguishing characteristic of these languages is the availability of constructs for describing intentional concepts: belief, intentions, desires, commitments, etc.
- Production rule languages. Production rule languages are a special kind of programming language, which is conceptually close to logic. They are one of the traditional methods of AI programming. They are not meant *per se* for specifying interactive systems, but, as we shall see, both 'intentional' concepts and interaction with the outside world show up in a natural way in some of the production rule based specification techniques.
- Task negotiation schemes. Although software systems incorporating task negotiation are relatively experimental, various such schemes are being proposed.

- Agent platforms. These are agent programming frameworks, usually focussed on practical interoperability and network communication. They typically provide software libraries such as internet communication wrappers, and schemes for routing and locating agents.
- Agent architectures. Agent architectures are models that prescribe how a complex agent should be built out of smaller components.
- Agent communication standards. Often-mentioned in the context of agents are the agent communication languages FIPA-ACL and KQML, which are attempts at worldwide communication standards. These languages should reflect a lot of the existing ideas on the subject of interactive systems, and will therefore be discussed in some detail.

2.4.1 Intentional logics

Intentional logics are general logical frameworks for describing agents' mental states in high-level terms, such as goals, intentions, and beliefs. In fact, some (Meyer, 1998) argue that such explicit modelling of intentionality is what truly characterises an agent system. Typically, the frameworks define extra sets of operators on top of some existing logical framework for describing time and events. Examples are BDI (Belief Desire Intention), based on temporal logic, and KARO (Knowledge Abilities Results Opportunities), based on dynamic logic.

Here are some operators from BDI (Veer et al., 1998, the sheets by J. J. Meyer):

$[]x$: from now on, x holds
$\Diamond x$: eventually, x will hold
$K_a x$: a knows that x holds
$B_a x$: a believes that x holds
$G_a x$: a has a private goal to achieve x
$I_a x$: a intends to achieve x

Using these operators, we might for example specify the rule:

$B_a I_b x \wedge B_a \neg \exists_{c \neq b} B_b G_c x \rightarrow B_a G_b x$

If a believes that b intends to achieve x, and it believes that b does not believe that there exists some other agent that wishes to achieve x, it will assume that b also has x as its private goal. In other words, if b doesn't seem to do x for some other agent, it probably does it for its own private purposes.

Discussion. The ideal situation is that an agent is completely described using such rules (but preferably, simpler ones from which other rules, such as the rule above, may be inferred). Theoretically, an agent's behaviour could be specified entirely by means of such rules. However (see also (Müller, 1996, section 2.3.3)):

- It is not easy to write down something meaningful using these kinds of 'advanced' logic. Even in specifying small systems, experts are known to make serious mistakes. A particularly infamous example is the Little Nell problem (Veer et al., 1998, J. Meyer's lecture). Other examples are the ill-definedness of some of the intentional frameworks that have been proposed (Baltag, 1999), and of the popular agent communication standards (van Eijk et al., 1999).
- Even the plainest form of logic required for reasonable expressiveness, predicate logic, is not executable, nor is verification of modal logics against computational behaviour computationally tractable (Wooldridge, 1998). The only computational feedback tools that remain are theorem provers and syntax checkers.

- The relation between the more expressive intentional logics and computer programs are not even well-defined, as pointed out in (Wooldridge, 1998). In particular, this includes the class of logics that enable modelling of knowledge about knowledge of other agents (and enable expressions such as the example above to be specified).

While intentional logics certainly are expressive, in the light of the other three requirements given in section 1.3, their usefulness as specification languages is questionable. Possibly, they are only useful as modelling languages, i.e. a language to model a well-behaved general framework. For use of such a framework as specification language, it will have to be filled in by a more concrete framework first.

In fact, it should be noted that, when people use the word 'BDI', they usually do not refer to BDI logic, but rather to any kind of explicit modelling of intentional concepts: see for example (Müller, 1996, section 2.3).

2.4.2 Production rule systems

In its most basic form, production rule specification is the complement of state automata specification: the ordering of events is not described explicitly. Instead, the state of the system is described as a number of facts, and the behaviour as a number of condition-action (or production) rules. Each condition is an expression in terms of facts, and each action may change some facts. If several conditions hold simultaneously, one may be chosen, perhaps according to some other criterion.

Production rules are found in expert systems. Example (Pachet, 1995):

```
decideNoTreatment
     "If pressure is normal then consider abandoning treatment"
     |Doctor d. Patient p|
             p bloodPressure < d maxBloodPressureFor: p.
             p hasTreatmentForTension.
     actions
             d considerAbandoningTreatmentFor: p.
```

Note that if the condition in a condition-action rule is not made false by the action, the rule will be triggered endlessly.

Goals and plans. A system often needs several actions to make a condition false. During the performance of the actions, facts may change in unexpected ways, possibly changing the course of actions to be taken, or initiating new actions. So, it should be possible to re-evaluate the facts during the execution of a sequence of actions. This means that the system's reaction to some facts should be split into several condition-action rules, and a processing state has to be maintained as part of the fact set.

In many production rule systems, this processing state takes the form of pending goals. Note that this corresponds to the 'flattest' interpretation of goals and intentions as found in intentional logic, without the complexities of recursively modelling the knowledge states of others. Modelling behaviour as a set or sequence of goals has a long history in both HCI (hierarchical task modelling) and software engineering (structured programming). Here is an illustrative example in CCT (Cognitive Complexity Theory), a formal HCI task modelling language (taken from (Haan et al., 1991)):

```
(PDELW1 IF (AND      (TEST-GOAL delete-word)
                (NOT (TEST-GOAL move cursor to %UT-HP %UT-VP))
                (NOT (TEST-CURSOR %UT-HP %UT-VP)))
        THEN ((ADD-GOAL move cursor to %UT-HP %UT-VP))
)
(PDELW2 IF (AND (TESTGOAL deleteword)
                (TESTCURSOR %UTHP %UTVP))
        THEN ((DOKEYSTROKE DEL)
              (DOKEYSTROKE SPACE)
              (DOKEYSTROKE ENTER)
              (WAIT)
              (DELETEGOAL deleteword)
              (UNBIND %UTHP %UTVP))
)
```

These two rules define how a user may delete a word in a text editor. The first rule defines that the cursor has to be placed on top of the word in case this was not yet the case. The second rule defines how a word below the cursor may be deleted. Unlike structured programs and plain hierarchical task models, these models enable arbitrary manipulation of the goal state. In this example, this is done by means of `ADD-GOAL` and `DELETE-GOAL`.

Pattern matching. Many production rule systems have some kind of pattern matching facility built in. In fact, it is already found in the CCT example above: in the first rule, the actual cursor position is matched with the goal cursor position. The following example from OPS-5 (Cooper and Wogrin, 1988) illustrates more clearly how it works:

```
(literalize monkey  at on holds)
(literalize object  name at weight on)
(literalize goal    status type object to)
...
(p mb7  (goal ^status active ^type holds ^object <w>)
        (object ^name <w> ^at <p> ^on floor)
        (monkey ^at <p> ^holds nil)
    -->
        (write (crlf) grab <w>)
        (modify 3 ^holds <w>)
        (modify 1 ^status satisfied))
...
```

The three `literalize` statements define three tuple data types `monkey`, `object`, and `goal`, with, respectively, length 3, 4, and 4. Any number of tuples may exist of each type. The set of all tuples comprise the set of facts of the system. Pattern matching consists of matching specific fields, usually for equality, across facts that are tested for in the condition part of a rule. If a match exists, the rule is triggered and the values of the fields of the match found may be used in the action part. In this example, the rule `mb7` specifies how a monkey picks up an object: if it has a goal to pick up some object `<w>`, and that object `<w>` happens to be in the same location `<p>` as the monkey, it is picked up, and the goal is marked as satisfied.

Pattern matching may be viewed as the ability to use 'SQL' type database queries in the condition part of a rule. The fact set has in fact the same form as found in SQL type

databases. This means that pattern matching may be done using a regular database query engine.

Interaction. An agent's interaction with the outside world may be modelled as facts that change spontaneously. Some models exist specifically for building multi-agent systems using production rules. A particularly interesting one is DESIRE (Jonker and Treur, 1998a). DESIRE is a hierarchical compositional model: each agent may in turn be composed of agents. The agents are connected by point-to-point channels, which have a fixed structure, though channels may be disabled and enabled while the system runs. Each agent may have one or several rule sets. Here is an example:

```
if     belief(at_position(food,P:POSITION),pos)
   and belief(at_position(screen, p0),neg)
   and belief(at_position(self,P:POSITION,neg)
then to_be_performed(goto(P:POSITION))

if     belief(at_position(food,P:POSITION),pos)
   and belief(at_position(self,P:POSITION),pos)
then to_be_performed(eat)
```

These rules specify the 'world interaction management' component of a mouse searching for food, given a fixed setup with a number of discrete locations and an impassable screen which may be in place or absent. The belief() facts are updated from another component, the 'world information maintenance' component, while the to_be_performed assertions are output again to other components. This example also illustrates the use of data types, which may be used to constrain pattern matching. `POSITION` is a data type specifying a specific subset of the set of facts. A data type may be part of a type hierarchy, which may be specified separately.

Active databases. The most naive implementation of a production rule system is an endless loop, in which all conditions are checked one by one. The system may check whether all conditions were false in the last iteration of the loop, and in which case it should go to sleep until an event from the outside world occurs that changes the facts. This way, the system is implicitly event-triggered. When there are many rules and facts, this method is inefficient, especially when the rules contain full-fledged database queries. A more subtle model would figure out which conditions might have changed given a certain change, and would only need to re-evaluate a subset of the rules.

An alternative is to specify for each rule, next to the usual condition, precisely one easily-identifiable type of event that triggers the rule. The subset of conditions which might be true as a result of the event is now immediately obvious. This is what active databases (Baral et al., 1999) (Kim, 1995) do: their conditions have the form

$event \wedge condition \rightarrow action$

This also means that, in their very simplest form,

$event_number \rightarrow conditional_procedure$

the rules may be implemented as a plain `switch...case`-type event handling loop. So, a plain event handling loop might be seen as the simplest type of production rule system.

Active databases usually allow regular updates to be made directly to the database. The rule system, specified separately, then tries to keep the database consistent. The regular updates are translated to events:

$update \wedge condition \rightarrow action$

or, in some systems,

$update_{start} \wedge condition_1 \rightarrow action_1$
$update_{end} \wedge condition_2 \rightarrow action_2$

$update_{start}$ denotes that an update is about to occur, and $update_{end}$ denotes that an update is finished. Example:

$add(client_order)_{begin} \wedge total_ordered + client_order > total_in_stock \rightarrow add(stock_order)$
$add(stock_order)_{begin} \wedge total_expenses + stock_order_expenses > total_income \rightarrow abort$

The first rule checks, whenever a client is trying to order a product, whether the amount ordered is actually in stock. In case it isn't, a new order is placed to re-fill the amount in stock. The second rule verifies, whenever a stock order is about to be placed, whether enough money will be available to pay for the order. For example, a client order which results in a stock order that cannot be paid is aborted.

Discussion. Production rule systems enable description of very complex behaviour, and have been successfully used in various AI applications. In fact, complex behaviour may be specified with only a few rules. However, systems with many rules easily become unmaintainable. For a description of some of the software maintenance issues, see (Cooper and Wogrin, 1988).

Production rule based specification may offer a complement to process algebra, enabling the more complex parts of a (prototype) agent to be specified more easily. It already has some successful history in HCI, showing that it is useable by other people than computer experts.

2.4.3 Task negotiation schemes

Providing general models for task negotiation is one of the most ambitious goals of multi-agent modelling. A few examples of recently-proposed task negotiation models are given here.

(Dignum et al., 1999) describes a model for team formation using intentional logic. The team is formed by a single initiator, which has a goal $\varphi = \varphi_1 \ldots \varphi_n$. The precise (temporal or data) relation between the actions φ_i is not addressed in detail. A number of potential team members are queried for their ability to do each of the individual actions. Then, the initiator tries to convince individual agents to cooperate in the goal by means of a persuasion dialogue, which is however not worked out in much detail. After a set of agents has been determined that is able to collectively achieve the goal, a broadcast signalling succesful task allocation may begin the achievement of the goal.

DECAF (John R. Graham, 1999) also uses a task hierarchy to specify a goal. The task is eventually composed out of atomic actions, which are programs written in any language. Each program has input and one or more specific outcomes, each with its own output. The data flow between the tasks and actions is specified by routes coupling one of the outputs to the input of another action or task. In fact, this model is just a dataflow model, in which each node has multiple output points, only one of which produces output for each input given.

In (Boella et al., 1999), tasks are specified as decomposable into sequences of subtasks called recipes, until only atomic subtasks remain. A task may have several recipes. Data is modelled as a world state which changes after each action has occurred. Note that this is similar to using global variables. Various properties of the atomic tasks, the world, and the existing agents, determine a global 'group' utility function that may be used by an agent to determine 'optimal' recipes for the tasks it wishes to achieve.

Discussion. Generally, tasks are specified as temporal orderings of atomic actions, similar to the usual hierarchical task models. Besides quality specifications such as duration and chance of success, the semantics of the atomic actions are generally not specified further in a way that is used by the system. The issue of managing the scope of, and the flow of data between, actions is often not addressed in detail.

While the overview given here remains very tentative, it may be said that the actual *negotiation* part of the task negotiation models–what makes the models multi-agent rather than 'regular' AI–is relatively immature (Meyer, 1998).

2.4.4 Agent platforms

Agent platforms (or agent operating systems) provide libraries and short-hands to make agent-based programming easier. Dozens of agent platforms have been proposed recently. They often model open and distributed systems, in which there is a dynamical number of agents, most of which will not know each other. They get to know each other by performing some lookup function. This lookup function is generally performed by a separate agent called a facilitator, a broker, or a matchmaker.

JATlite. (Petrie, 1996) This is apparently one of the most popular agent platforms, and has been used for some real-life applications. The system is composed of five layers. An agent may use one or more layers at will. From bottom to top:

- Abstract layer: describes an abstract API specification, providing low-level communication concepts such as address, connection, message buffer, and the agent communication module itself, providing higher-level methods for reacting to received messages, and sending messages.
- Base layer: implementation of the abstract layer using TCP.
- KQML layer: provides basic support for parsing, generating, sending, and receiving inter-agent messages. The messages are specified in KQML. See section 2.4.6 for a detailed description of KQML.
- Router layer: provides a routing and message queueing mechanism, using separate routing agents called Agent Message Routers (AMR). In fact, it is much a re-invention of the Email service, which already provides location lookup, message routing, queueing, and relay.
- Protocol layer: provides an API for accessing existing application-layer Internet protocols, such as FTP, HTTP, SMTP, and POP3. Oddly, this layer is located at the top, rather than below or next to the KQML layer, as one might expect.

Except for the router functionality, JATlite mostly provides wrappers around standard internet services. No task negotiation, planning, or knowledge functionality is provided.

KAoS. A KAoS agent exists within an environment called an agent domain. There is a domain manager, which keeps track of the existence of all agents within the domain. Proxy agents must be used to communicate to agents outside of the domain. There is also a matchmaker agent, which is used to obtain agents' identities, given a capability description. The way capabilities are or should be specified is not described.

The possible communication protocols between agents are specified using finite-state automata. Each agent plays a specific role in the protocol. Each protocol is part of a coherent

set of protocols, called a suite. The specification of protocols, roles, and suites may be used to specify the requirements on an agent's capabilities to participate in specific agent conversations.

Discussion. These two models are typical for the agent platforms found. Some provide only internet facilities, which sometimes includes process migration. Others provide facilities for formal protocol specification and agent lookup services. Facilities for modelling physical domains and routing mechanisms may be considered too low-level for our purposes, and, in case they prove necessary, they may be delegated to a lower layer. The most interesting new feature of agent platforms is the agent lookup service.

Agent lookup is generally performed by a special type of agent called a facilitator, matchmaker, or broker. A facilitator can find an agent, given a description of properties the agent should have. The facilitator is in most of these models the most important means to obtain identities of new agents. The property description given to the facilitator is usually just a name. If we assume that the name is the name of a protocol (similar to a module's or an object class' interface), the facilitator is conceptually very close to a dynamic linker, such as Unix's `ld` or Java's `ClassLoader`.

Typically, multiple facilitators are allowed, each keeping a specific database of agents. It may be said that a facilitator stands for a specific context. For example, some agent platforms are explicitly hard-coded for discrete locations (i.e. computers), with agents that reside in them, and possibly, migrate between them, somewhat similar to autonomous robots. Inside each location is one facilitator, which keeps track of the agents in that location. Next to modelling a physical context, the location also defines a conceptual context. Consider the following issue: how does a program, as it is started up, know which screen it should display its windows to? This sounds like a nonsensical issue, but it becomes less nonsensical in a distributed environment.

For example, assume there is a distributed system with several GUI agents, one for each user. Now, assume that some agent wishes to find an agent to do the task 'display window'. This task is only meaningful if the window is displayed on the right screen. So, it is not possible to assign the task to *any* GUI agent, but rather, to a GUI agent that has access to the right screen. The task could be made more specific: 'display window to a specific user'. Another, inflexible and un-agent-like, option is to send the request directly to the right agent, rather than to the agent community in general, assuming that this agent is somehow known beforehand. Yet another option is to restrict the request to a specific context, such as the user's machine, making the user implicit. In some of the agent platforms, it can clearly be seen that the physical context does double duty: it addresses a mixture of implementation and conceptual issues. For example, in Sodabot (Coen, 1994), as well as in the older version of JATLite (JATLite 0.3), the user context, bound to a specific physical location, also uniquely defines access to the GUI. The physical context (the computer) is used to constrain the meaningful actions of an agent, somewhat like autonomous robot systems. However, with such a scheme, it is not possible to distinguish several GUIs on the same machine, or several GUIs per user, while such a thing is already possible with traditional systems. To increase flexibility and conceptual clarity, it may be a good idea to separate the two concepts, or perhaps do away with the physical context altogether, and let another layer manage physical issues.

Contexts do not need to be disjoint. Possibly, facilitator re-directions may be cascaded, i.e. one asks a global facilitator for a local facilitator, and the local facilitator for a specific agent. There might be various meaningful contexts within a system, which may or may not overlap. For example, concerning a user in a VE, we could have:

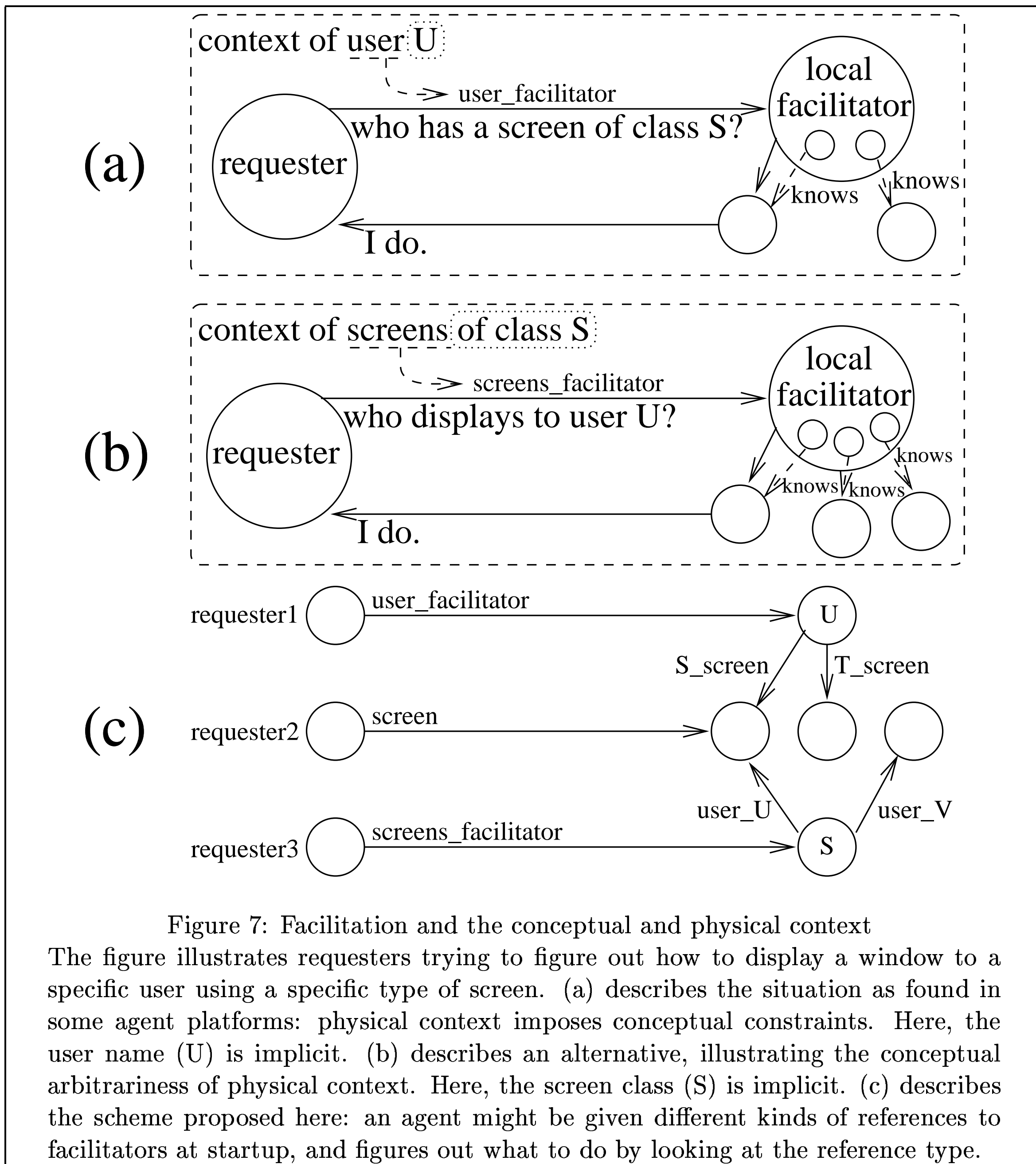

Figure 7: Facilitation and the conceptual and physical context
The figure illustrates requesters trying to figure out how to display a window to a specific user using a specific type of screen. (a) describes the situation as found in some agent platforms: physical context imposes conceptual constraints. Here, the user name (U) is implicit. (b) describes an alternative, illustrating the conceptual arbitrariness of physical context. Here, the screen class (S) is implicit. (c) describes the scheme proposed here: an agent might be given different kinds of references to facilitators at startup, and figures out what to do by looking at the reference type.

1. VE context, containing all objects or agents that have a physical presence in the VE. Within this context may be a visual context, in other words, what the user sees from where s/he is situated in the VE.
2. user interface context, containing the windows a user is interacting with. Possibly, the user may be using several screens simultaneously.
3. conversation context, containing the agents the user is currently conversing with.
4. dialogue context, containing the objects (for example, the performances) that are are or have been in topic during the current conversation.
5. physical context, containing everything related to one computer: its screen, keyboard, speakers, disks, etc.

One way to view the agent system proposed below is to see each agent as a facilitator. Different kinds of facilitators, standing for different kinds of contexts, are distinguished by labelling the reference to each facilitator with a type name. Basic facilitation and lookup schemes may be described in the system in a straightforward manner. See figure 7.

2.4.5 Agent architectures

Agent architectures provide standard libraries and compositional guidelines for building complex agents out of simpler ones, emphasising the macro aspects of agent design. Typically, the simpler agents are special-purpose agents, each addressing a specific aspect in the functioning of the more complex agent, such as reaction to the environment, long-term planning, social interaction, and maintaining a world model. For this reason, such complex agents are sometimes called hybrid agents. An overview of agent architectures can be found in (Müller, 1996, section 2.6).

A well-known agent architecture is INTERRAP (Müller, 1996). An INTERRAP agent has three layers: behaviour-based layer, local planning layer, and cooperative planning layer. Each layer is an agent with specific capabilities. INTERRAP includes software libraries specifically meant for each type of agent.

Another architecture is offered as part of the compositional guidelines in the DESIRE model (see also section 2.4.2). A DESIRE agent is composed of 7 modules: agent-specific task module, agent interaction management, maintenance of agent information, world interaction management, maintenance of world information, own process control, and cooperation management.

Discussion. The basic compositional structures of these two systems are very similar. In DESIRE, the same three layers are found as in INTERRAP, except that the bottom two layers are each split into two separate modules. Only the 'agent specific task module', which may be used to 'configure' an agent without changing the other modules, is not found in INTERRAP.

The INTERRAP architecture actually provides relatively detailed and workable standard libraries for development of a complex agent. Some applications have been built using it, for example, COSMA (Busemann et al., 1997).

2.4.6 Agent communication language standards

Both KQML (Knowledge Query and Manipulation Language) and FIPA-ACL (the ACL from the Foundation for Intelligent Physical Agents) are attempts at international agent commu-

nication language standards. In both languages, each message has a type, indicating the 'illocutionary meaning' (for example, request or inform) of each message. Each message type has a number of valid types of reply. Each message has a contents section in which the body of the message should be specified in some (textual) format. With each message, the name of the content's language and some ontology (which is, in practice, a sort of vocabulary) should be specified. In FIPA-ACL, the name of a protocol determining which messages may follow this one may also be specified separately.

The languages are, necessarily, general frameworks, leaving most details open. Roughly, they are generalisations of existing interoperable message-passing systems, such as email, with the notable addition of message types. Example of an `inform` message in FIPA-ACL:

```
(inform
  :sender      agent1
  :receiver    auction-server
  :content     (price bid good02) 150)
  :in-reply-to round-4
  :reply-with  bid04
  :language    sl
  :ontology    auction )
```

Though message types may be defined at will, the bulk of the documents specifying the two languages describe a number of pre-defined message types, suggesting some frequent uses. FIPA-ACL also specifies some formal requirements that each message type should have on an agent's knowledge state, but, as noted before, the semantics remain ill-defined (van Eijk et al., 1999). It also specifies four primitive messages (inform, request, confirm, disconfirm) out of which all messages should be built. Below is an overview of the available types (note that this classification partially follows the one given in the KQML standard).

1. **Action negotiation**

   Unlike the other types of performatives, ACL is more elaborate than KQML here. It has specific performatives for refusing an action and action failure, and several types of action request.

   ACL offers:

propose	propose an action
cfp	call for action proposals
accept-proposal, agree	accept some proposal to perform an action
reject-proposal	reject some proposal
request	request to do an action
refuse	refuse to do an action
cancel	request an agent to stop an action
failure	inform that action failed

   KQML offers:

achieve	request to do actions necessary to make content true
unachieve	request to undo the effects of a previous achieve

2. **Knowledge inform**

   In KQML, it is possible to give a command to directly manipulate another's knowledge base, as opposed to simply supplying information. Note that the way in which KQML's performatives are stated (especially delete-one versus delete-all) suggests that an agent has a database, rather than knowledge base.

   ACL offers:
   inform
   confirm
   disconfirm

KQML offers:

tell	content is in sender's knowledge base
untell	content is not found in sender's knowledge base
deny	the negation of content is in sender's knowledge base
insert	ask for insertion of content
uninsert	ask for undo of previous insert
delete-one	ask to delete one sentence matching query
delete-all	ask to delete all sentences matching query
undelete	ask for undo of previous delete

3. **Knowledge query**

ACL offers:

inform-if	when sent inside a request body, it is a question
inform-ref	when sent inside a request body, it is a question asking for an object with a given description
query-if	ask for truth of specific proposition
query-ref	ask for object with given description

KQML offers:

ask-if	ask for existence of specific content
ask-all	ask for all variable instantiations for which given content is true
ask-one	ask for one variable instantiation for which given content is true

4. **Error signalling**

ACL offers:
not-understood

KQML offers:

error	signal error in a previous message
sorry	signal that agent understands but is unable to process some previous message

5. **Automatic notification, requests over time, and managing continous streams of data**

Both KQML and ACL have performatives requesting for updates whenever an agent's knowledge changes. ACL offers requests to do actions as a result of knowledge changes, rather than just informing. KQML's stream-all enables a query result to be answered in a stream of messages. KQML also offers some general stream control performatives.

ACL offers:

request-when	request to do action when certain condition becomes true
request-whenever	request to do action every time a condition becomes true
subscribe	ask for object with given description, and for all subsequent changes in that object

KQML offers:

subscribe	ask for all updates to the response of the given performative
stream-all	as ask-all, but with one message per instantiation
eos	signal the end of a stream of messages
standby	ask to signal the readiness to respond to the given message
ready	signal readiness to respond to some previous message
next	ask for next response to some previous message
rest	ask for the rest of the responses to some previous message
discard	discard all further responses to some previous message

6. **Finding other agents**

In both languages, there are specific agents (in KQML, they are called facilitators) that can be queried to find other agents. Basically, the facilitator enables messages to be forwarded to the agents that it knows about. In KQML, there are performatives for registering with a facilitator. It is unclear how this is done in ACL.

ACL offers:

propagate	a message containing an agent property description and propagation constraints is sent to an agent which is assumed to know about the existence of other agents. The message may be propagated further
proxy	request to send message to all agents which have a specific property
request-whomever	unclear, there's a bug in the documentation here
KQML offers:	
advertise	advertise a capability to another agent. When sent to a facilitator, advertise a capability to all agents.
unadvertise	cancel advertise
register	register one's name and additional information with a facilitator
unregister	cancel previous register. Also cancels all advertises made to the facilitator.
transport-address	announce a change of address
forward	forward message to given agent
broadcast	forward message to all agents that the receiver knows of
broker-one	ask for one response to given message from any agent the receiver knows
broker-all	ask for all responses to given message from all agents the receiver knows
recommend-one	ask for one agent that is able to respond to given message
recommend-all	ask for all agents that are able to respond to given message
recruit-one	make one agent respond to given message
recruit-all	make all agents that are able to respond to given message respond

Discussion. These languages essentially provide:

- A general message syntax in system-independent (text) format. However, the syntax of the most involved part, the content of the message, has been mostly left open.
- A set of suggestions for often-used message types. The precise meaning of the message types is oddly underspecified. While, for example, many of the agent platforms are advertised as supporting either of these languages (though KQML appears to be the most popular by far), in retrospect it remains unclear what this actually means. Some of them only support message syntax, which is not very helpful, because that is the most trivial part of the languages.
- Some facilities for message sequence protocols. Protocols have been specified for the suggested message types, though these are, again, somewhat underspecified. FIPA-ACL also provides a separate protocol field.

The standards are necessarily general and are still underspecified, and almost everything still has to be filled in. The standards do include suites of message types for dealing with most of the aspects given in section 2, including update reactions and agent lookup services.

2.4.7 Discussion

It is perhaps interesting and somewhat distressing to note that the compositional issues of many of the agent-based models are largely under-emphasised. When looking at many of the toy systems, one wonders how these could or should be scaled up to full size.

Techniques, or even examples, that show how large software systems may be composed out of agents, are largely absent, though some work is finally being done on that (Wooldridge et al., 1998). What kind of concurrency problems will occur? How will the necessity of handling these kinds of complex communication protocols and knowledge bases turn out? Just how coarse should the granularity of an agent system be? Techniques and examples for specifying and classifying actions and capabilities for use with task negotiation or facilitation are also hard to find. Are action hierarchies (similar to structured programming) a good way to specify

operations, given the recognised weaknesses of structured programming with respect to data dependencies? How does one go about documenting and maintaining action specifications?

Even though many of the intentional logics, task negotiation schemes, and ACLs may have something to offer for dialogue system and VE development, their relative immaturity makes their immediate applicability questionable. The first agent model proposal described here will be mostly based on interactor models and agent platforms, incorporating the most interesting features of these. A 'micro level' specification language may be developed as part of the model, which may be based on production rule specification. Possibly, useful features of agent architectures may be incorporated on top of the existing model where needed.

3 Proposal of agent system

This system is an attempt to specify an update notification model that encompasses and generalises upon the concepts found in existing models. Update notification is one of the more basic and arguably more promising features of the models described. The main idea of the agent system proposed here is that the system as a whole is viewed as a data structure, of which the agents may have some local awareness. This is similar to interactor models, in particular Amulet, as described in section 2.3, and to the facilitator model described in section 2.4.7. Instead of limiting ourselves to a tree data structure, as some interactor models do, the more general case of a directed labelled graph (i.e. an entity-relation diagram) is used. Like Amulet, agents may specify queries about the graph structure relative to their own position in the graph. Agents may have properties, which have values that may be queried by other agents. Unlike Amulet, agents may also send messages directly through one of the arcs that it is connected with: they are like references in the OO model. Unlike OO references, the communication may proceed is a symmetrical way: both parties may initiate communication. The arcs will be called relations.

3.1 First prototype

The feasibility of this idea was tested by building a prototype with example application that is simple enough to implement in a few days. After this quick feasibility test, a second, more stable and elegant, prototype was built, taking about a week to complete. Further extensions may be implemented or adopted as needed. The prototypes are implemented in Java using Remote Method Invocation (RMI) to provide distributed communication. The agents may be run from a standard web browser. The current architecture is described below.

At each moment in time, each agent i may have a number of properties and be involved in a number of relations:

Properties:	$P(i, value)$,	$P \in Properties$
Relations:	$R(i, j, value_i, value_j)$,	$R \in Relations, j \in Agents$
	$R(j, i, value_j, value_i)$,	$R \in Relations, j \in Agents$

In this description, sending messages through a relation is modelled as modification of relation attributes. Relation attributes are a common construct in entity-relation modelling. Each agent has an attribute to which it may make changes, which will be notified to the other agent. Relations are similar to references in the OO model. As in OO and process algebra, it would be easy to define protocols for each of the relations, determining which values or value types are allowed, and in what sequence. Such protocols could be formally described using finite-state automata or process algebraic specifications. Other kinds of relations, such as ternary relations or relations without value could be modelled as well, if the need arises.

The basic communication model chosen in this first version is a buffered model, rather than a synchronous one as provided by process algebras. This is for practical reasons: now, a process does not have to wait for other processes to handle its messages, and the number of network synchronisations decreases, improving performance significantly. Other models may be emulated using the buffered model.

Each agent has a private part, which controls the agent's behaviour. It may change, create, and delete properties at will. Relations may only be established with the consent of the other party. It may also initiate queries. Queries are updating queries, that is, they remain active until terminated explicitly, and all changes to the initial query result will be automatically notified. See figure 8.

If we view relations and attempts to establish relations as just another kind of query, a query

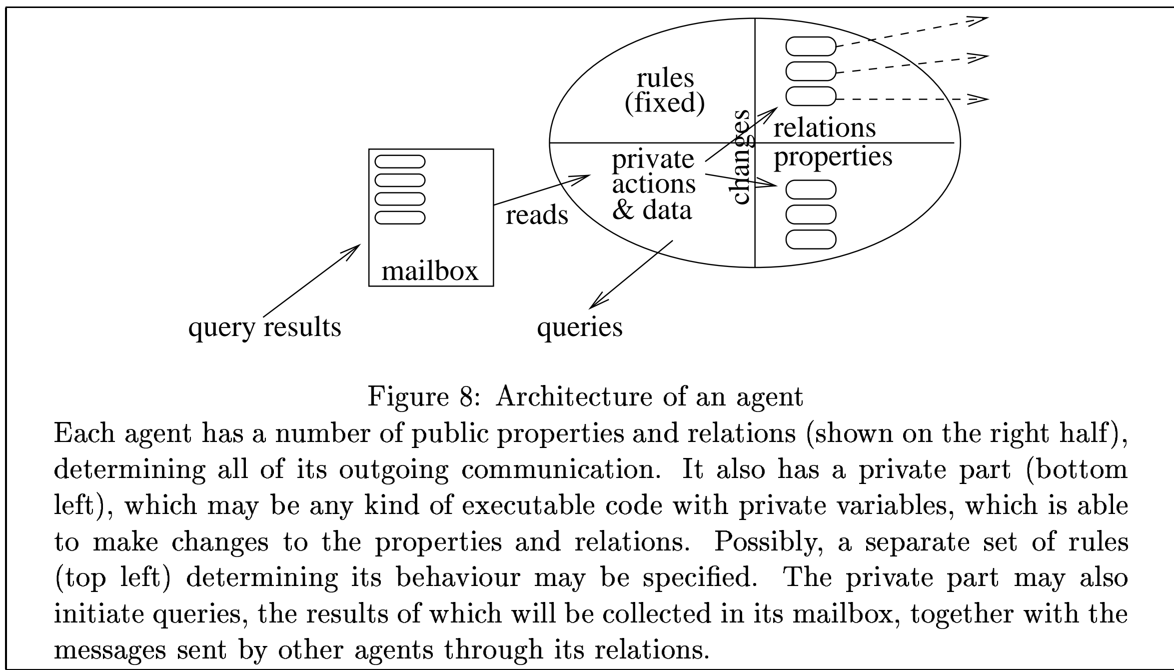

Figure 8: Architecture of an agent

Each agent has a number of public properties and relations (shown on the right half), determining all of its outgoing communication. It also has a private part (bottom left), which may be any kind of executable code with private variables, which is able to make changes to the properties and relations. Possibly, a separate set of rules (top left) determining its behaviour may be specified. The private part may also initiate queries, the results of which will be collected in its mailbox, together with the messages sent by other agents through its relations.

has the form:

```
<QUERY> = <SETQUERY> | <PROPQUERY> | <RELQUERY> | <LISTENQUERY>

<SETQUERY> =   <RELATION> . server ( <SETQUERY> )
             | <RELATION> . client ( <SETQUERY> )
             | self

<PROPQUERY> =  <PROPERTY> . has    ( <SETQUERY> )

<RELQUERY>  =   <RELATION> . request ( <AGENT> )
              | <RELATION> . provide ( <AGENT> )

<LISTENQUERY> =   <RELATION> . requesters ( )
                | <RELATION> . providers  ( )
```

with `<RELATION>` the name of a relation, `<PROPERTY>` the name of a property, and `<AGENT>` an agent ID. Note that the syntax precisely corresponds to Java statements. `<SETQUERY>` and `<PROPQUERY>` define the regular queries, querying the graph structure. Each such query takes as its parameters a relation name and a set, and results in a set. If we assume agent i is doing the queries:

`<SETQUERY>`: $\begin{cases} R.server(X) & = \{y | R(x,y) \wedge x \in X\} \\ R.client(Y) & = \{x | R(x,y) \wedge y \in Y\} \\ self & = \{i\} \end{cases}$

`<PROPQUERY>`: $P.has(X) = \{(x,v) | P(x,v) \wedge x \in X\}$

Note that the relations' values may not be queried by third parties. More complex queries may be provided in future versions. For example, a set join and intersection would be useful.

Note, also, that the terms 'server' and 'client' are used for the two participants in a relation. Though this suggests an asymmetry, both parties have exactly the same capabilities at their disposal. The two roles are, however, distinct. In other words, $R(i,j)$ is not the same as

$R(j,i)$. In case $R(i,j)$ (which may alternatively be written iRj) holds, i is called the server, and j is called the client. The terms 'agens' and 'patiens', or 'subject' and 'object' might be used as alternatives to 'server' and 'client'.

`<RELQUERY>` defines the queries for establishing relations, taking as parameters the relation name and the other party's ID. As soon as a party i has started a query `relation.request(j)` and the party j has started `relation.provide(i)`, then the relation is established, and each party receives an acknowledgement message. The queries may then be used to receive messages from the other party. The order in which the request and provide queries are issued is not important.

An agent should have some way to know whether others are requesting relations with it. This is what `<LISTENQUERY>` is for. The agent IDs of the requesters or providers will be passed as results to this query.

The notifications of each change in a property or relation is sent immediately to the relevant mailboxes. This way, all changes arrive at the queriers in the order they were made, and the views the parties obtain of the situation at the moment of reading their mailbox is always up to date.

As a useful programming feature, each query may be assigned an arbitrary identity number. This number is then assigned to all messages that came from that query, as they are added to the mailbox. This way, the mailbox's messages may be pre-sorted easily by means of a simple switch-case construct.

In the current prototype, the queries are handled by a central server, which only accepts one change at a time. This is an easy way to ensure that all information remains consistent and up to date, and all update messages are queued in the order in which the actual events happened. A more distributed implementation is a subject of future research. The server maintains the existence of relations and the values of properties. There is also a more practical advantage of having one central server: only the computer running the server needs to run a RMI registry. The clients only need standard Java facilities.

A small example application has been built with the prototype, see figure 10. It is a small VE with a room in which any agents may enter, exit, move, and talk to each other. It is quite easy to have multiple rooms, though no standard scheme has yet been implemented for easy moving between rooms. Each agent may look around by querying the room's inhabitants through the contains relation. An agent may speak and move, by updating its lastsaid, pos, and shape properties. Each object in the room may have a brain which controls its actions. If it does not have a brain, it remains passive, but it may still be picked up and dragged around. Picking up is done by establishing a holds relation. The object automatically follows the position of its holder. Points_at relations may be established also, enabling any agent to keep track of what objects another agent is pointing at. Currently, there are two types of brain: a user brain, which enables an object to be controlled through a user interface, and an 'AI' brain, which contains part of the Schisma system, and is able to parse sentences related to theatre performances. The AI reacts to any user who comes close to it. It is quite easy to have both multiple users and multiple Schisma agents in the system.

3.1.1 Discussion

The example application is meant to indicate whether this model works and has practical value, and what further practical improvements should be made. Some observations are given in this section.

The nature of updating queries makes the amount of data being passed relatively small. Only

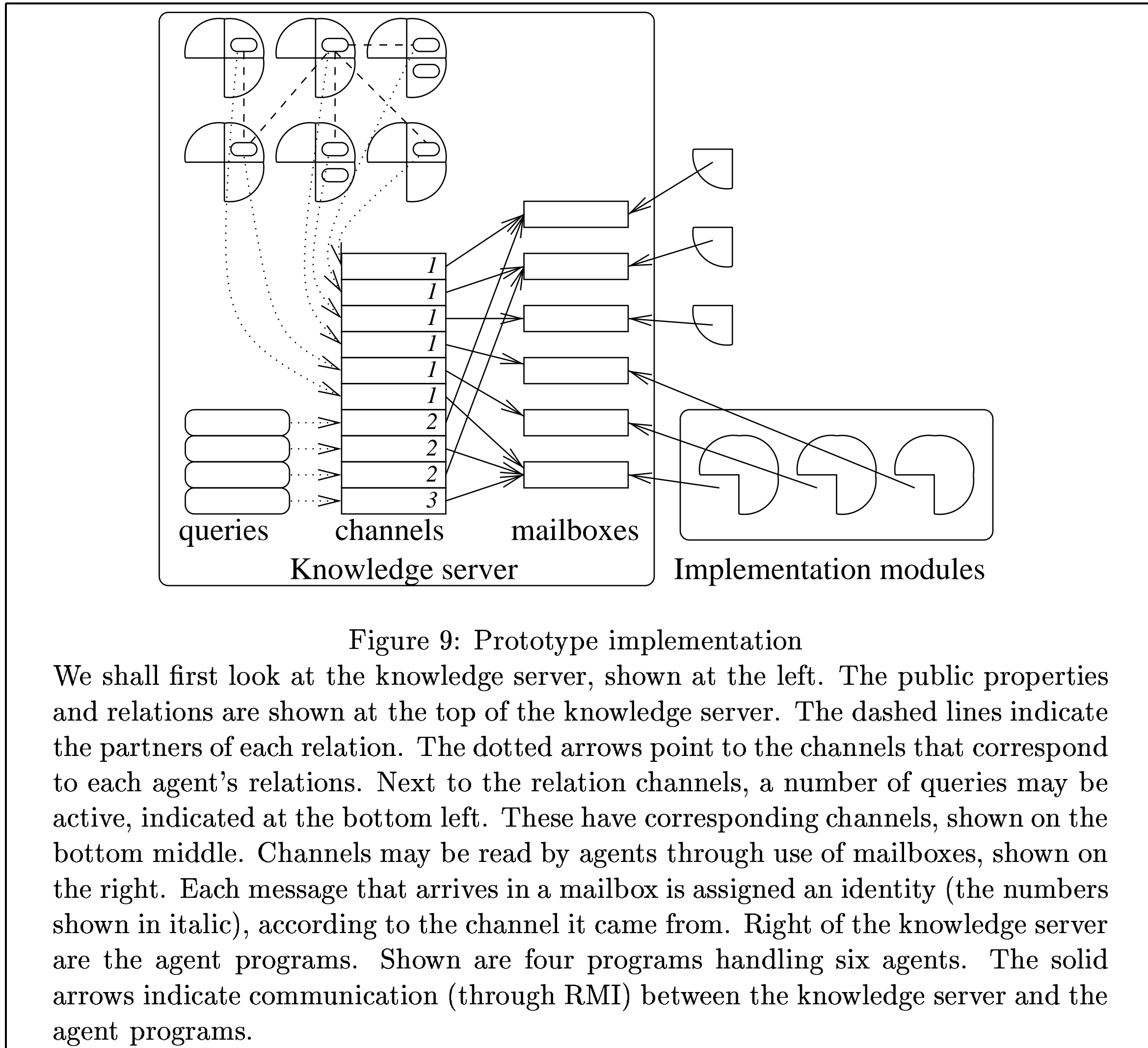

Figure 9: Prototype implementation

We shall first look at the knowledge server, shown at the left. The public properties and relations are shown at the top of the knowledge server. The dashed lines indicate the partners of each relation. The dotted arrows point to the channels that correspond to each agent's relations. Next to the relation channels, a number of queries may be active, indicated at the bottom left. These have corresponding channels, shown on the bottom middle. Channels may be read by agents through use of mailboxes, shown on the right. Each message that arrives in a mailbox is assigned an identity (the numbers shown in italic), according to the channel it came from. Right of the knowledge server are the agent programs. Shown are four programs handling six agents. The solid arrows indicate communication (through RMI) between the knowledge server and the agent programs.

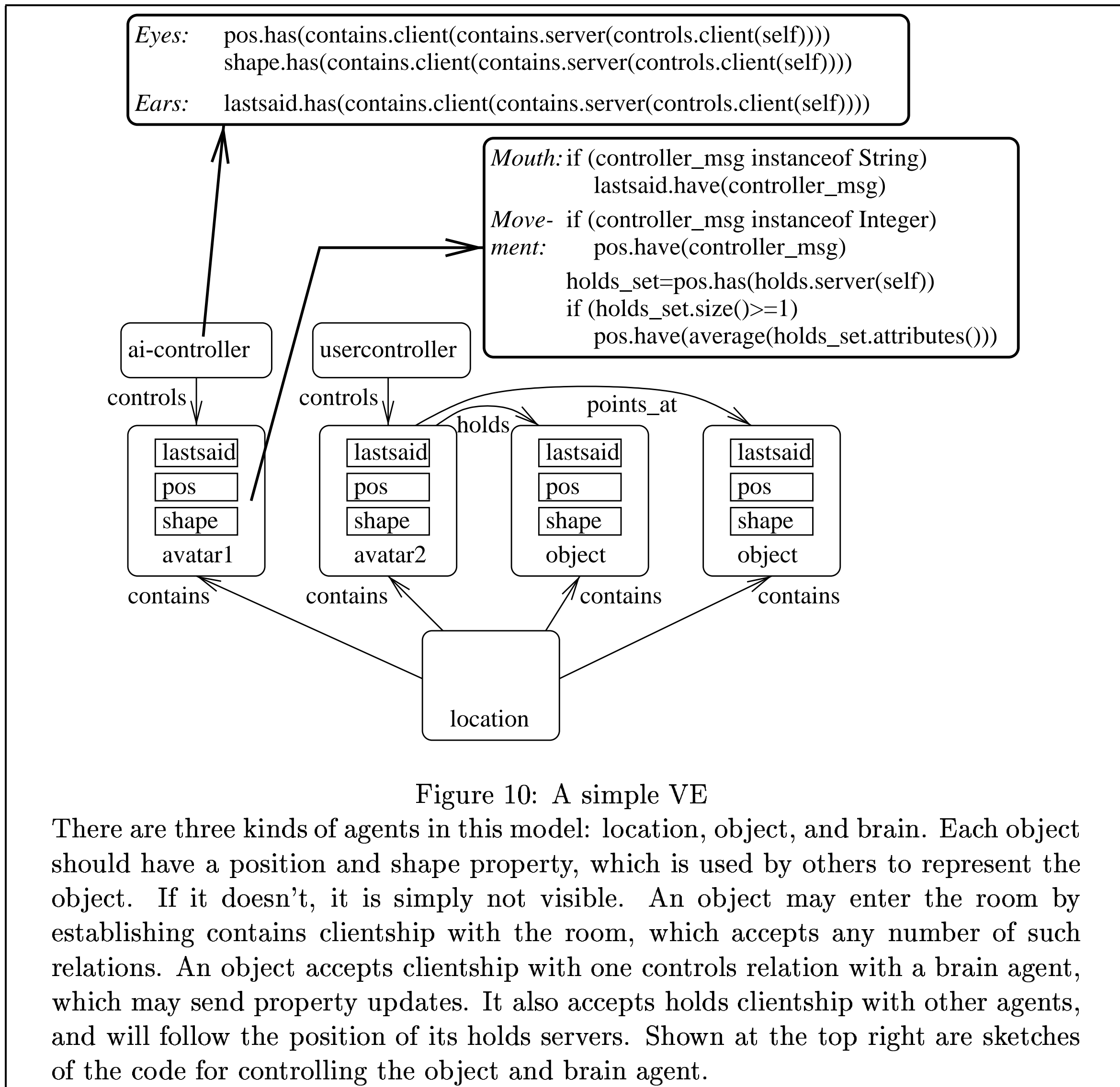

Figure 10: A simple VE

There are three kinds of agents in this model: location, object, and brain. Each object should have a position and shape property, which is used by others to represent the object. If it doesn't, it is simply not visible. An object may enter the room by establishing contains clientship with the room, which accepts any number of such relations. An object accepts clientship with one controls relation with a brain agent, which may send property updates. It also accepts holds clientship with other agents, and will follow the position of its holds servers. Shown at the top right are sketches of the code for controlling the object and brain agent.

when entering a room does an object receive all positions and shapes of its inhabitants. From then on, only individual changes are notified. If an agent only changes position, only its new position will be passed to the other agents. This model would be well suited for VEs with more complicated shapes, such as VRML descriptions: these would only be loaded once under normal circumstances, yet any changes an object makes in its VRML description afterwards would still be notified.

The buffered model ensures that the amount of network (RMI) calls are limited. A typical VE agent only needs to fetch its messages when it wishes to update its display. When multiple agents are moving around, each receives a list of changes once every time it updates, rather than multiple single changes during its update. As it turns out, the performance overhead of RMI calls is relatively large, as is, probably, the overhead of network synchronisations in general. So, the buffered model provides quite an improvement in efficiency over the unbuffered model. Though the agents currently have to read their mailboxes through a 'polling' scheme, the ability to block when the mailbox is empty will be provided in future versions.

Changing properties, on the other hand, has a relatively large overhead: currently, there is one RMI call per changed property. A scheme in which property changes could be queued, and then committed with a singe call, would significantly increase network performance. Alternatively, the problem could be worked around by putting the information that is changed most often in a single property.

Specifying agent behaviour turned out to be relatively easy. Concurrency, information delay, and failure handling turned out to be less of a problem than expected: relatively few bugs emerged during the specification of the agents. While the actual interaction in the example remains very simple, it may be expected that much of the object-agent interaction within a VE will not be much more complex. At the same time, the model should be able to provide the basic communication and facilitation functionality for modelling more 'intelligent' agent-agent interaction, which will probably be highly domain-specific or experimental anyway. The agent programs could be specified using a reasonably simple program structure: the queries and initial relations are initialised once at the beginning of the program, while a main loop consisting of a `sleep - execute plan - handle messages` loop turned out to be sufficient for the agents in the example application. Pre-sorting messages by means of a switch-case on the 'channel identity' number works well, and provides a readable program structure.

A negative aspect of the model is its verbosity in some places. In the current 'raw' Java agents, there is various 'red tape' code strewn across the program, such as a separate list of numbers specifying the channel identities, and a separate list of variables for keeping track of query results. Sending and handling messages through a relation is more verbose than doing a method call: in case of sending, failure must be handled, and in case of receiving, the message type must be checked first. Also, unlike obtaining a reference in OO, establishing a relation requires the requester to wait for acceptance. A special (relatively simple, macro type) language to automatically generate code for most of these cases is under development.

The limitations of the still rather limited queries were likely to cause problems, even for this first application. So, some extensions that have already been found to be missing are discussed here. The extensions that were already mentioned are set intersection and union. However, these are not strictly necessary. These can be performed afterwards, by having a separate query on each individual set, and performing the operations afterwards. The ability of adding such set operations directly to the query format would only be an efficiency issue. A limitation that is more serious are the 'flat' sets that the queries always result in. For example, if one would like to know what all agents in a room are holding, one would have to query each agent's holds clients in turn. This bypasses the rule of only querying with oneself

as the only absolute reference point. What is worse, the agent cannot set up a single query but must create or delete queries when agents enter or leave the room. This introduces much more verbosity in the code, and the possibility of the different query results running out of sync. While it is possible to specify a query resulting in the set of all objects that all agents in a room hold, the result does not tell who is holding what. A couple of additional operators (something like `has_servers` and `has_clients`) resulting in sets of sets would likely be a great improvement. Another option is to adopt an existing database engine that supports updating queries and is very efficient for simple queries.

3.2 Conclusions and future research

An attempt at a relatively complete overview of interactive systems models has been made. The overview is meant to document previous research, and provide a necessary and hopefully sufficient rationale for the new model proposed here. The overview necessarily remains sketchy and fragmented in some parts, and possibly contains serious gaps. Naturally, the overview may be filled in further, as needed by future developments.

The proposed model has been shown to work for a simple VE example, warranting further investigation. The very first thing to do is build a larger example in order to further test the feasibility of the model. Currently, a 3D version of the VE application is being developed using Java3D, incorporating the original VMC VRML environment. The rest of the section describes several aspects of the proposed model that may be developed further.

Agent specification language. As argued before, it would be useful to provide some standard macros and shorthands, instead of programming in raw Java. Furthermore, the language may be structured similar to a specification language other than a plain procedural programming language. Since the agent programs are typically event handling loops, a 'production rule' or 'active database' type language seems appropriate. In fact, considering our relatively detailed evaluation of process algebras, OO, and intentional logics, a more detailed account of production rule specification would be appropriate at this point.

Documenting and checking agent programs. An agent model which allows complex dynamics requires techniques for documenting and (model-) checking agents' behaviour. Already mentioned is the possibility of specifying a protocol for each channel, describing which sequences of values are allowed. This could be described naturally using finite-state automata. Next to this, sequences of combinations of relations could be defined. Each agent may have specified how many of each kind of relation it may have, and which combinations or sequences are allowed.

Furthermore, it would be useful to be able to model-check the internal behaviour description of an agent. For the complex mix of production rule type specification and Java code that is likely to be used, only part of the specification is likely to be checkable. Some trade-off between expressivity, checkability, and abstractness will have to be made, which will depend on further practical experience.

Further refinement of the model's capabilities. The setup of the model has in part been determined by ease of implementation. Various extensions come to mind: already mentioned were query extensions. Different ways to communicate could be added, for example, the ability to send a message directly to an agent, without having to go through a create-relation / delete-relation cycle. The listener-type queries could be extended, enabling an agent

to specify beforehand what kind of agents it will accept relations with. The relations could be ordered in some type hierarchy, similar to objects in the OO model. This way, more generic or specific queries could be specified. Access restrictions could be placed on each relation and property as well.

Relieving the requirement of a central knowledge server The central knowledge server is an easy way to circumvent concurrency problems with handling query updates. A more distributed, and in fact, more natural model would be to have each agent maintain its own part of the knowledge locally. In such a model, the 'facilitator' concept is followed more literally. In order to ensure that all information remains up-to-date, each change could be synchronised with all parties concerned with the change. In particular, each modification of a property or relation should be synchronised with all queriers. To be able to achieve this, each agent should be aware which agents are observing its information. The creation of new relations also means that both parties transfer query information associated with the relation. Synchronisation may be achieved by communication through existing relations. Other possibilities to implement a more distributed model exist, for example, having one knowledge server per computer, and having the different knowledge servers only communicate when information crosses the border between the knowledge domain of one server and another. Possibly, the requirement of having up-to-date and in-order information at all times will need to be relaxed.

4 Appendix: a developer's guide

This section is a first attempt at a developer's guide, and should already be useful to get a 'feel' for the system for those who are further interested in practical development. The guide consists of an API overview and a small guidebook.

4.1 Overview of API

An agent participating in the agent system needs to deal with eight special object classes. Four of these are the RMI objects used for inter-agent communication, and one is the message object. These are described below. Two are the Agent and Channel objects, which are just identifier values. The last one is the AttrSet class, which incorporates a set datatype, in which each set element may have an additional attribute.

AgentRegistry. The first thing any agent does is contact the AgentRegistry by means of a RMI lookup. Here, it obtains its identity and mailbox. For convenience, an agent may also look up other agents by name using the registry.

void	del_agent(Agent agent)	Delete agent.
Mailbox	get_mailbox(Agent agent)	Get mailbox of agent.
Agent	lookup_agent(String name)	Look up agent ID by name.
Agent	new_agent()	Obtain a unique ID for a new agent.
Agent	new_agent(String name)	Obtain a unique ID for a new named agent.

Mailbox. All messages arrive through channels, but may only be read by the agents through the mailbox. Once a channel identity is obtained, the channel may be added to the mailbox. Each channel may be given a separate type identity `chantype`, which is set in any message that is received through this channel, so that the message's origin may be easily distinguished.

void	add_channel(int chantype, Channel chan)	Add channel's future messages to the mailbox.
void	del_channel(Channel chan)	Delete given channel from mailbox and close the channel.
Message[]	fetch(boolean block)	Fetch all messages from mailbox.

Property and Relation. The knowledge may be accessed and modified though RMI objects of the Property and Relation class. All methods setting up queries return channel identities, standing for channels through which the various query result messages are passed. Note that the channels may in turn be used as an input for another query, enabling more complex queries to be made. It is currently not allowed to use a channel as input for multiple queries, or to add a channel that is already used for query input to the mailbox.

Property:

Channel Channel	has(Agent agent) has(Channel agents)	Test an agent set specification for the property and obtain its value.
void	have(Agent agent, Object value)	Assert that agent has this property, or that the value of the property changed.
void	unhave(Agent agent)	Assert that agent does not have this property.

Relation:

Channel Channel	server(Agent agent) server(Channel agents)	Define a new query which queries the set of agents that serve any of the agents in the supplied query.
Channel Channel	client(Agent agent) client(Channel agents)	Define a new query which queries the set of agents that are client of any of the agents in the supplied query.
Channel Channel	provide(Agent recipient, Agent agent) request(Agent recipient, Agent agent)	Request server- or clientship to the given agent, and set up a channel on which all events related to further development of this relation may be monitored.
Channel Channel	providers(Agent recipient) requesters(Agent recipient)	Create a channel that can be used to monitor any requests for server- or clientship with this agent.
void void	serversend(Agent sender, Agent recipient, Message message) clientsend(Agent sender, Agent recipient, Message message)	Enables one party of the relation to send a message to the other party.
void void	serverrefuse(Agent sender, Agent agent) clientrefuse(Agent sender, Agent agent)	Deletes relation in a neat manner, refuse a request or provide, or delete own request or provide.

Message. The Message class incorporates the information passed through channels. A list of constants for use with the message type field are also specified here. A message has the following fields:

int	type	The message type.
int	chantype	The channel type that the message originated from. Undefined if the message was not read through a mailbox.
Object[]	content	The message's actual content.
Agent	sender	The sender identity, which is null unless stated otherwise (see below).
long	timestamp	A timestamp, which is set when the message first passes through a channel.

A message may have one of the following types. Other types may be defined as needed, preferably by subclassing the Message class.

Query set message types. In all cases, the content is an set with attributes (class AttrSet).	
InSet	The set of agents which satisfies the query. This message is sent only once, at the moment that query is set up.
EnterSet	A set of agents that should be added to the original query result.
LeaveSet	A set of agents that should be deleted from the original query result.
Messages indicating transitions in the relation state. In all relation messages, the sender field is set to the participant on the other side.	
NotListening	Warning that the other party does not have a requesters/providers query on this relation.
Accept	The other party accepts the relation.
Refuse	The other party refuses.
Abort	The relation is aborted.
Miscellaneous message types	
Request	requesters/providers query message type. The sender field is set to the requesting/providing agent.
ChanClosed	'Meta' message, which is sent just before a channel is closed.
Any	'User defined' type message, may have any content.

The lifecycle of a relation between any pair of agents, the methods that lead from one state to another, and the messages that are generated as a result, are given in the state diagram in figure 11. There are four main states: nonexistent, requesting, providing, and established. Note that a serverrefuse / clientrefuse call may be used at any point to revert the relation's state to nonexistent.

4.2 A small design guidebook

Because this model appears to be a relatively new way to design VEs, some preliminary design ideas and guidelines will be given here, illustrating how it may be used. Note that, as yet, this guidebook only addresses macro issues, since the single-agent specification technique is still under development.

Implementing agents in software modules. The conceptual relation between agents and software modules need not be one-to-one, see figure 12. An agent program may be a simple wrapper communicating privately to one or more software modules, or to one or more independent data structures inside a software module, such as separate files on the same file server or separate windows on the same screen. In this case, the modules or data may be viewed as part of the agent. Even a user who controls an agent may be viewed as part of the agent.

It is also possible to have one module handle several agents. In fact, if the agent system requires a regular entity-relationship database, its entities may be modelled as agents. The rest of the agents need not be aware which program handles which agents: it all appears as one large data structure to them, accessed in a uniform way. If the database is not important enough, or too large or complex, it may still be modelled using a single agent which is queried through message passing.

Maintaining views. It is common practice to model 'physical' GUI and VE objects as separate modules. In this model, such objects, or possibly, coherent groups of such objects, would be modelled as agents. The screen on which the agent is represented may be either:

- implicit, so that it always displays on the same screen, and is supposed to maintain its

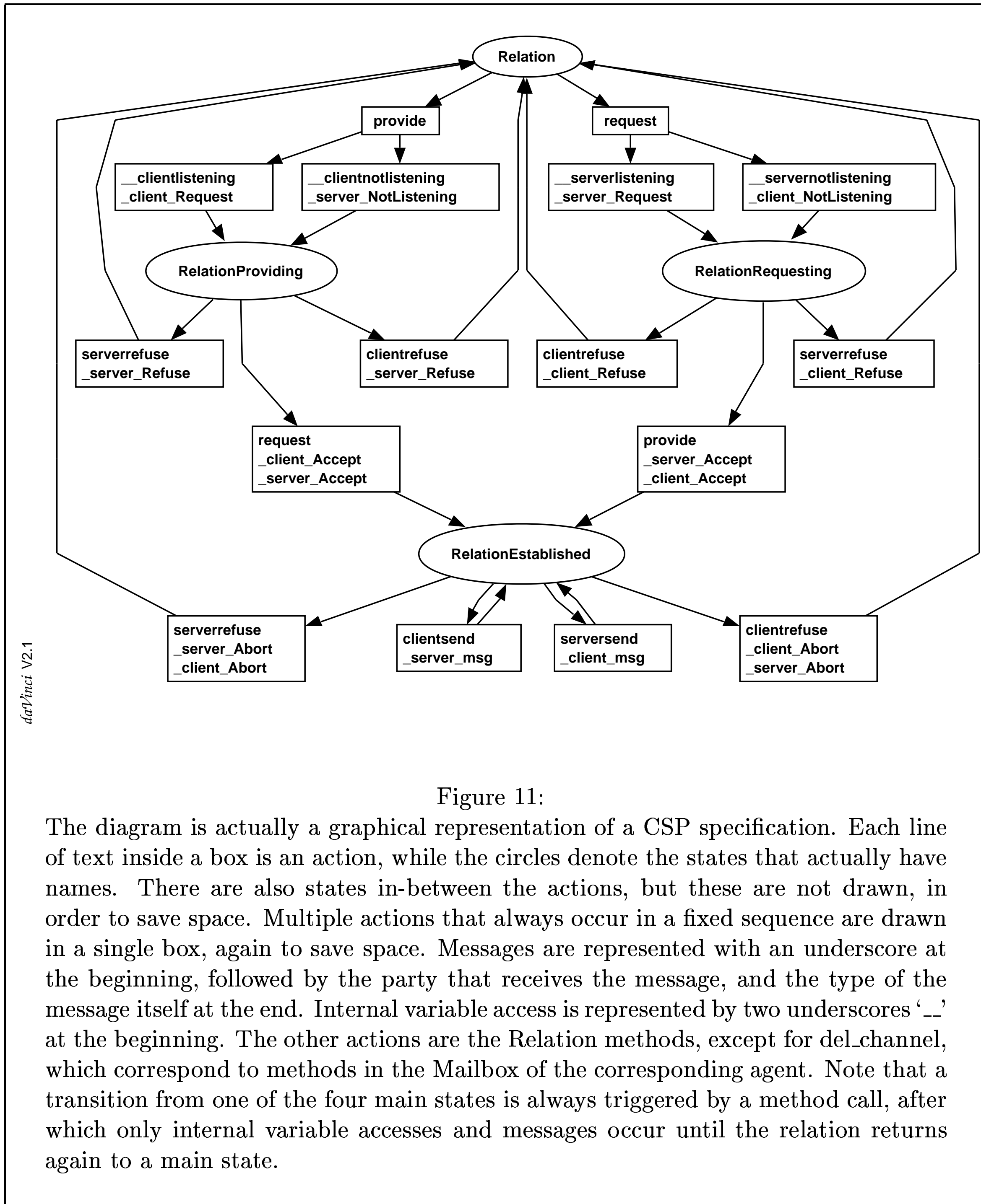

Figure 11:
The diagram is actually a graphical representation of a CSP specification. Each line of text inside a box is an action, while the circles denote the states that actually have names. There are also states in-between the actions, but these are not drawn, in order to save space. Multiple actions that always occur in a fixed sequence are drawn in a single box, again to save space. Messages are represented with an underscore at the beginning, followed by the party that receives the message, and the type of the message itself at the end. Internal variable access is represented by two underscores '__' at the beginning. The other actions are the Relation methods, except for del_channel, which correspond to methods in the Mailbox of the corresponding agent. Note that a transition from one of the four main states is always triggered by a method call, after which only internal variable accesses and messages occur until the relation returns again to a main state.

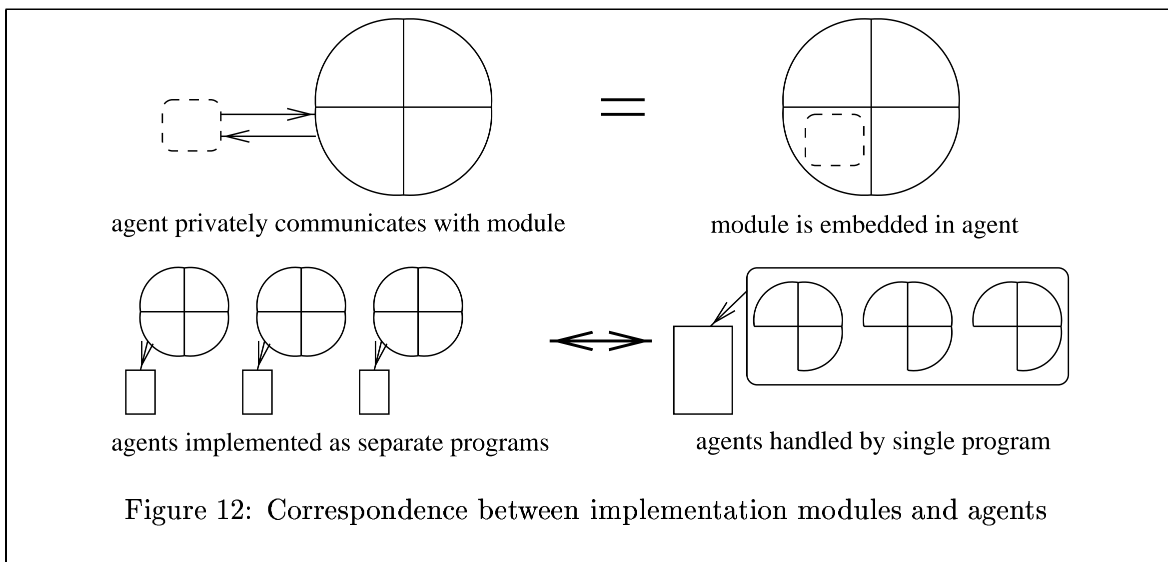

Figure 12: Correspondence between implementation modules and agents

own graphical representation. The object has private communication with the screen. This is a good choice for GUI objects which are only ever observed by a single user.

- explicit, so that it may be displayed on different screens, depending on its relations. The object should have either properties that (abstractly) describe its appearance, or communicate its appearance through some of its relations. This is a good choice for multi-user objects, such as objects in a multi-user VE.

If the VE may contain multiple autonomous agents and/or users, an obvious choice is to model the agents and users as separate objects in the VE, i.e. avatars. Each user's input and output may be viewed as embedded in his/her avatar. The user's view may be maintained by the avatar querying its environment. Computer avatars have the same possibilities at their disposal, but will probably find the relational structure of their environment more useful than the graphical representations.

Modelling a physical world. Instead of maintaining a highly-detailed physical model of the VE, some physical constraints between objects may be modelled using relations, with the additional advantage of having a semantic representation that is more readily useable by computer agents. For example, an object being on top of another may be modelled using an on_top relation. In fact, the well-known SHRDLU system (Winograd, 1972) already uses a similar model to model the scene that is manipulated by the SHRDLU AI: the stacking of objects is modelled explicitly using a `SUPPORTS` relation (the reverse of on_top). When an agent tries to pick up an object, the object may easily be made to refuse when it sees that it is already client of an on_top relation. When the bottom object moves, the top object may easily be made to move with it. When one object is placed on another, the other object may accept or refuse, depending on its shape. For example, the sphericalness of an object may be modelled by the object refusing all on_top requests. Other examples are rooms bordering on other rooms, possibly separated by doors, rooms and objects inside or attached to other objects, and avatars holding objects.

Highly complex physical relations between agents may be handled by a separate constraint agent, which checks the constraints, and sends results to the agents concerned. For example, the room in which agents are located is a natural place for a collision, proximity, and hearing range checker. This approach is similar to the one used in the VRML constraint programming system described in (Richard and Codognet, 1998) and (Richard et al., 1998). Here, real-time

constraints are maintained by having script-nodes send their updates to a constraint node, which sends back correction commands as soon as constraints are violated.

More high-level semantic information that is not strictly physical may be modelled in the graph structure. For example, a user pointing at an object may be modelled using a points_at relation. The most obvious advantage is that computer agents may easily keep track of a user's pointing behaviour. Pointing might also be made observable by other users, for example, by having the objects highlight themselves when they are being pointed at. Other examples are manipulation and visibility.

Modelling inter-agent write access. Write access from one agent to another may proceed through establishing a relation. The relation could either imply the querying of a specific property of the writer by the reader, or the values written could be communicated through the relation. Dependencies that other agents have on the newly-written values may be modelled using communication, or by a property having the value, so that it may be queried by the other agents.

Several agents may attempt to write to the same data. Concurrency problems may be circumvented by only allowing one agent to establish the write relation at any time. Starvation problems may be circumvented by having each agent break the write relation as soon as possible.

As an example, consider a multi-user text editor with scrollbars example, similar to the example found in (Calvary et al., 1997). In this system, multiple users simultaneously edit the same text. Each user has an edit window, through which changes may be made. The panning position of the edit window of each user can be observed by the others, so that the users are aware of each other's location.

Figure 13 shows an agent-based version of this system. It is assumed that the panning position may be changed by one or more GUI components, for example a scrollbar and a 'jump' button that enables saving the current position and restoring a saved position. There are two kinds of concurrent access: users accessing a document and components accessing a panning position. In both cases, the other users/components should be notified of the changes. In this example, writing is modelled as an `updates` relation. The reading agents may read the written values by simply having a query `<PROPERTY>.has(updates.server(self))` open at all times, and storing the results in their appropriate properties. The agents observing the changes may do so by means of a `text.has(displays.server(self))` and `pos.has(positions.server(self))`. If the text becomes too large, an incremental update scheme may be implemented through communications through the `displays` and `updates` relations. The users may observe each other by means of a query `<PROPERTY> . has(displays.server(displays.client(self)))`. It should be possible to have an arbitrary number of users and positioning components by simply allowing each document and window to have an arbitrary number of `displays` and `positions` servers.

Modelling fluid animation. In a multi-agent VE, multiple agents, possibly running on different computers, may animate concurrently. Shape and position updates may proceed asynchronously, as they will be automatically notified to the other agents. This allows each agent to run at its own frame rate. If the animation of agents running at particularly low frame rates appears jerky, some interpolation scheme could be used. For more detailed information on interpolation issues, see for example (Ryan and Sharkey, 1998).

This asynchronous scheme contrasts with the parallel scheme (as described in Reactive-C) that is often used for concurrent animation. The asynchronous scheme may however be

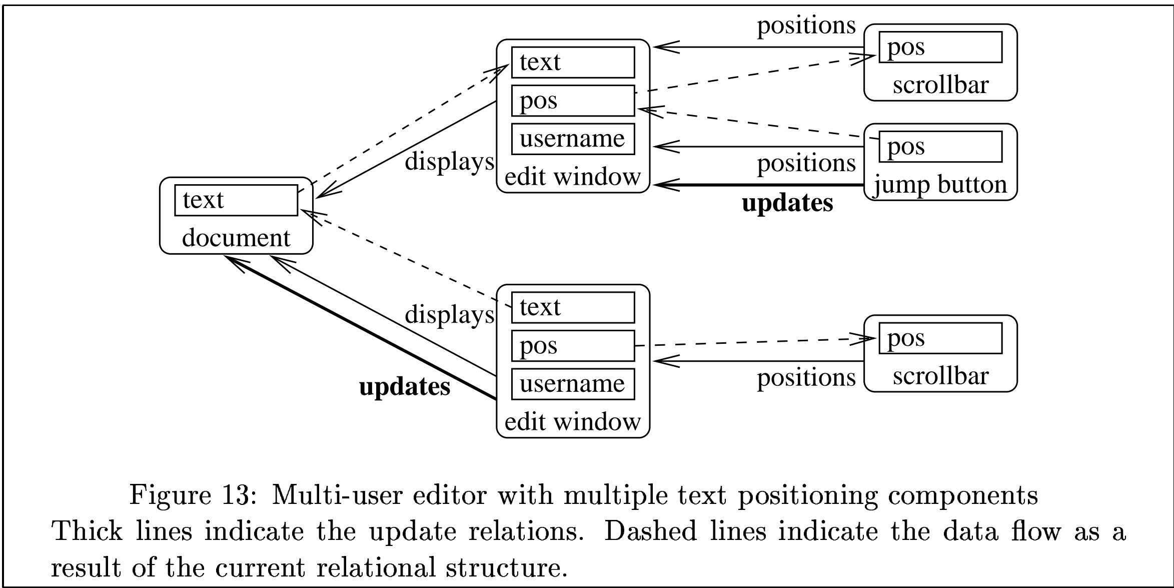

Figure 13: Multi-user editor with multiple text positioning components
Thick lines indicate the update relations. Dashed lines indicate the data flow as a result of the current relational structure.

quite acceptable for objects moving around independently, and has the advantage of not requiring central synchronisation. However, a problem occurs if the animation of two agents is interdependent. For example, consider two blocks stacked onto each other. If the bottom block moves, the top one should move with it. The top block could keep a query on the bottom one's position, but its position would then lag behind, making the animation less convincing.

Two schemes for addressing this problem are offered here, see figure 14. One possibility is to make the top block have a relative position, which is used instead of its regular position for the duration of the `on_top_of` relation. This way, the blocks always move together, and the relative position may even be changed independently. As soon as the relation is broken, the observers should revert to the regular position. Another possibility is to emulate the (Reactive-C style) synchronous model using a local sync node for the two blocks. The sync node has a `is_stable` property, which may be used by observers to determine when the value updates they received are valid.

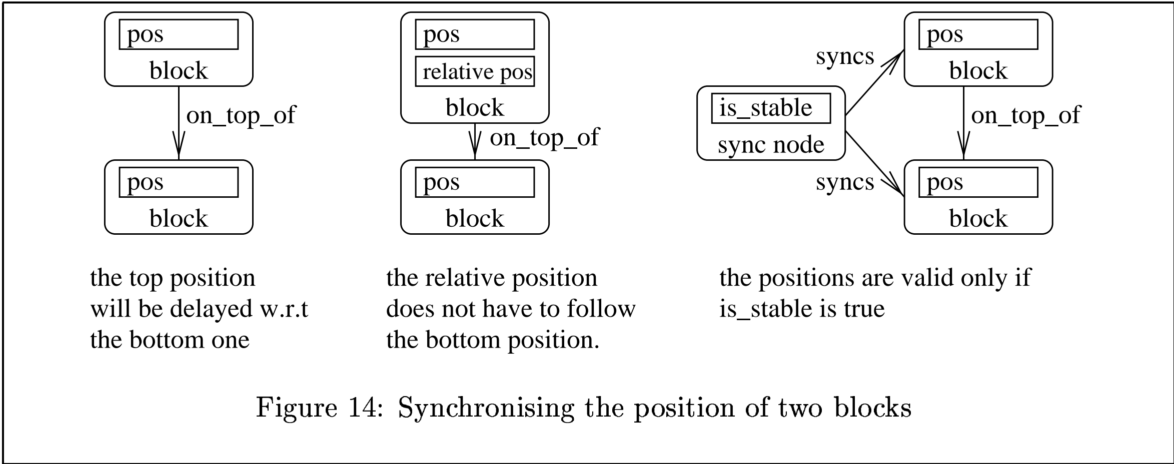

Figure 14: Synchronising the position of two blocks

4.2.1 Examples

To conclude this small 'guidebook', two sketches of larger example applications are given, see figure 15 and 16.

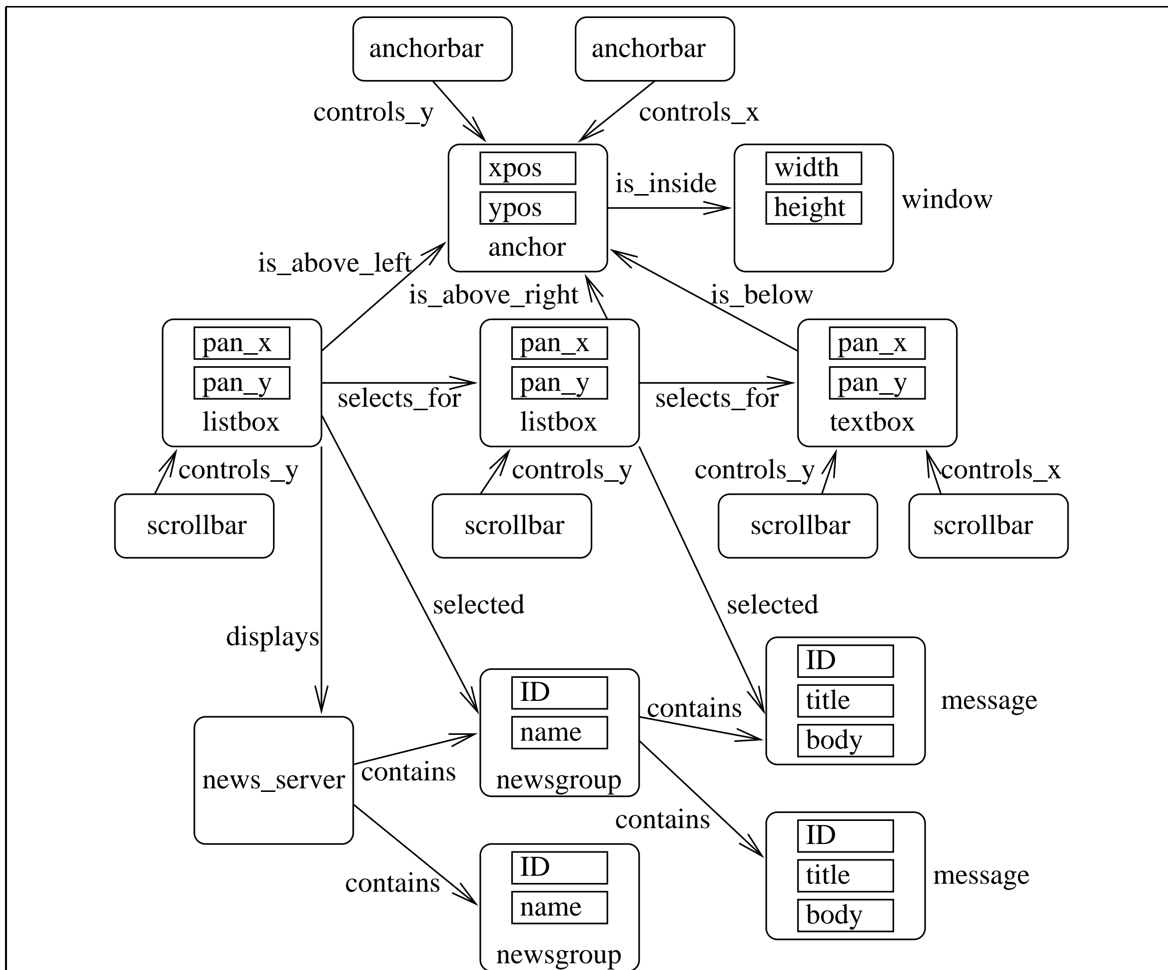

Figure 15: A News agency

This example is similar to the Netscape example given in figure 1. Note that the anchorbars, used to resize the windows, are also modelled here. The system is modelled using only six standard GUI components. It should be easy to model other applications, for example, a mail reader, using the same components in a similar setup.

Note that the window is a separate agent here, specifying the area in which all windows should be contained. There are a number of relations explicitly specifying the positioning relations between the windows, since the positions and sizes are interdependent, and may be modified. The `anchor` component in the middle specifies the point at which the windows meet each other. The windows are specified relative to the anchor according to the `is_...` relations. The two listboxes and the textbox use either the relations `displays` or `selects_for...selected` to determine their contents. The listboxes expect these to lead to an agent which has `contains` relations with the items to be displayed, and pick up any displayable properties these items have. The textbox expect these to lead to an agent with a `body` property.

The actual news database is modelled using a hierarchy of agents, with a news server agent at the top. These agents may be handled by a wrapper around the actual news server. This wrapper may be implemented only to load news items on demand, for example, only when a `displays` or `selected` relation is established.

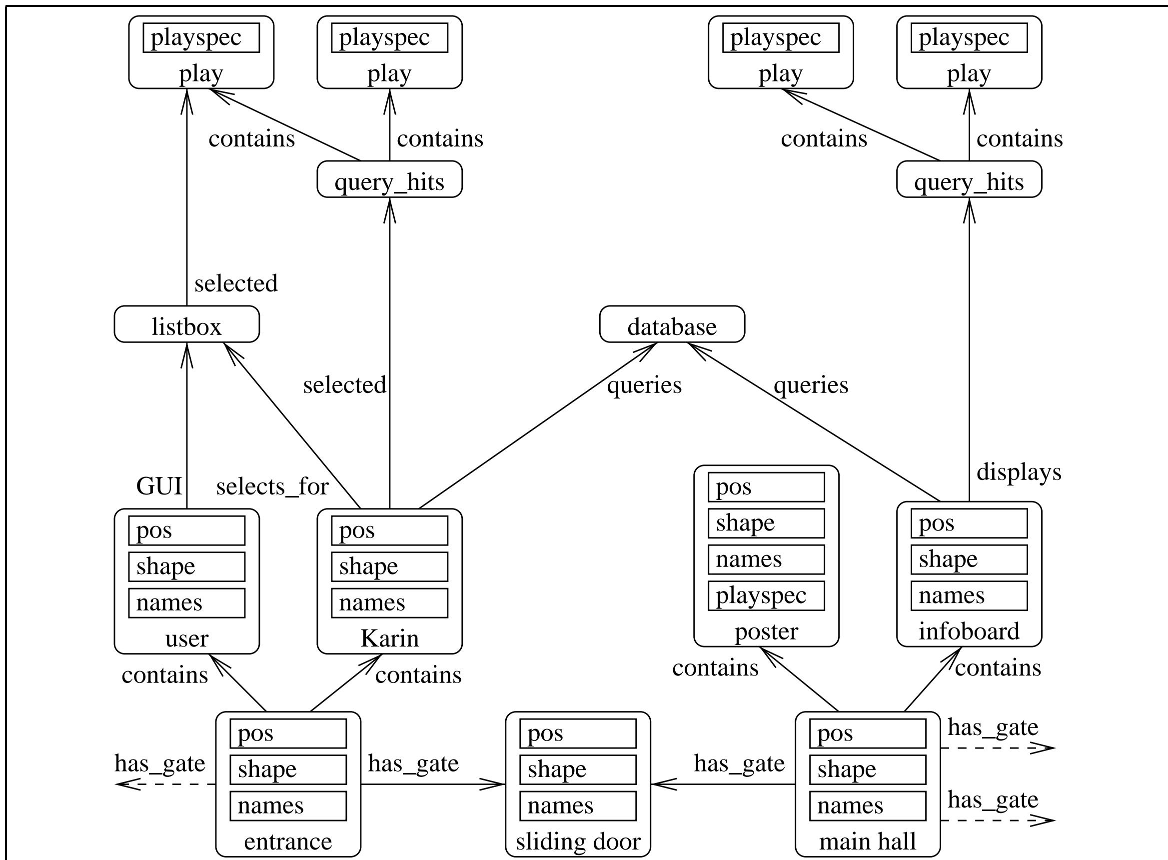

Figure 16: A VMC agency

The figure shows a sketch of an agent model of the VMC. At the bottom is the structure of the music centre, which are modelled using a web of room agents connected by gates, which determine the physical areas through which the rooms may be navigated or viewed. Using such a structure, rather than just a single room agent, may have several advantages: one is that the size of the site may become arbitrarily large, since clients only need to load a part of the whole site at any time, and the locations could be distributed across several machines. Another is that a navigation helper agent may use this structure to help a user find a route to a specific place. In the ideal case of the locations being all convex in shape, the VE may be convincingly navigated by walking straight from gate to gate.

Above the rooms are the VE agents. The information board is conceptually the same as the listbox, using a `displays` relation to determine what it shows. Karin maintains a dialogue with the nearest user, and maintains a `selected` relation with the latest query results. A listbox, shown privately on the user's screen, may display these last hits. The user may point at the information board or click in the listbox, causing a `selected` relation with the corresponding play. A similar relation may be established when the user points at the poster. It should be quite easy for Karin to observe this pointing and clicking behaviour, and use it to resolve linguistic references.

Both Karin and the information board use the database indicated by the `queries` relation. After specifying a query, this database may return a query_hits agent related to the query results.

5 References

References

Agarwal, R., Sinha, A. P., and Tanniru, M. (1996). The role of prior experience and task characteristics in object-oriented modeling: an empirical study. *International journal of human-computer studies*, 45:639–667.

Baltag, A. (1999). A logic of epistemic actions. In van der Hoek, W., Meyer, J.-J., and Witteveen, C., editors, *ESSLLI99 workshop: Foundations and applications of collective agent based systems (CABS).*

Baral, C., Lobo, J., and Scherl, R. (1999). A logical approach to building agents, active databases and workflows: representing and reasoning about actions. Lecture notes from the ESSLLI99 course.

Bernsen, N. O., Dybkjaer, H., and Dybkjaer, L. (1998). *Designing interactive speech systems: from first ideas to user testing.* Springer Verlag.

Boella, G., Damiano, R., and Lesmo, L. (1999). A utility based approach to cooperation among agents. In van der Hoek, W., Meyer, J.-J., and Witteveen, C., editors, *ESSLLI99 workshop: Foundations and applications of collective agent based systems (CABS).*

Bolognesi, T. and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59.

Booch, G. (1994). *Object-oriented analysis and design, second edition.* Benjamin/Cummings.

Boussinot, F., Susini, J.-F., and Hazard, L. (1998). Distributed reactive machines. Technical Report 3376, INRIA, France.

Brazier, F., Cornelissen, F., Gustavsson, R., Jonker, C. M., Lindeberg, O., Polak, B., and Treur, J. (1998). Agents negotiating for load balancing of electricity use. In *Proceedings of the 18th international conference on distributed computing systems (ICDCS'98)*, pages 622–629.

Brun, P. and Beaudouin-Lafon, M. (1995). A taxonomy and evaluation of formalisms for the specification of interactive systems. In *People and computers X: Proceedings of the HIC'95 conference.*

Busemann, S., Declerck, T., Diagne, A. K., Dini, L., Klein, J., and Schmeier, S. (1997). Natural language dialogue service for appointment scheduling agents. In *Fifth Conference on Applied Natural Language Processing*, pages 25–32.

Calvary, G., Coutaz, J., and Nigay, L. (1997). From single-user architectural design to PAC*: a generic software architecture model for CSCW. In *CHI '97.*

Carey, R. and Bell, G. (1997). *The Annotated VRML 2.0 Reference Manual, 1st Edition.* Addison-Wesley.

Coen, M. (1994). SodaBot: A software agent environment and construction system. Technical Report 1493, MIT AI Lab, U.S.A.

Cooper, T. A. and Wogrin, N. (1988). *Rule-based programming with OPS5.* Morgan Kaufmann publishers.

Dahl, O. J., Myhrhaug, B., and Nygaard, K. (1968). *Simula 67 : common base language.* Norwegian Computing Center.

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. (1995). The effect of inheritance on the maintainability of object-oriented software: an empirical study. In *Proceedings of the 1995 international conference on software maintenance.*

Davies, S. P. (1993). Expertise and display-based strategies in computer programming. In *People and computers VIII: proceedings of the HCI '93 conference.*

Dignum, F., Dunin-Keplicz, B., and Verbrugge, R. (1999). Dialogue in team formation: a formal approach. In van der Hoek, W., Meyer, J.-J., and Witteveen, C., editors, *ESSLLI99 workshop: Foundations and applications of collective agent based systems (CABS).*

Donk, O., van Dijk, B., and Nijholt, A. (1999). U-WISH, specification techniques for multimodal dialogues. Technical Report TR-CTIT-99-13, University of Twente, Centre for Telematics and Information Technology.

Fischer, C. (1997). Csp-oz: A combination of Object-Z and CSP. Technical Report TRCF-97-2, University of Oldenburg.

Gibbon, D., Moore, R., and Winski, R. (1997). *Handbook of Standards and Resources for Spoken Language Systems.* Mouton de Gruyter, Berlin.

Goldberg, A. and Robson, D. (1989). *Smalltalk-80, the language.* Addison-Wesley.

Green, T. (1990). Programming languages as information structures. In *Psychology of programming (computers and people series)*, pages 118–137.

Gurr, C. A. (1994). Supporting formal reasoning for safety-critical systems. *High Integrity Systems*, 1(4):385–396.

Haan, G. d., Veer, G. C. v., and Vliet, J. C. v. (1991). Formal modelling techniques in human-computer interaction. *Acta Psychologica*, 78(1-3):27–67.

Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice Hall, New York.

Holzmann, G. J. (1991). *Design and validation of computer protocols.* Prentice Hall.

Hussey, A. and Carrington, D. (1996). Using Object-Z to compare the MVC and PAC architectures. In Roast, C. R. and Siddiqi, J. I., editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface.*

John R. Graham, K. S. D. (1999). Towards a distributed, environment-centered agent framework. In *ATAL99: Sixth International Workshop on agent theories, architectures, and languages.*

Jonker, C. M. and Treur, J. (1998a). Agent-based simulation of reactive, pro-active and social animal behaviour. In *Proceedings of the 11th international conference on industrial and engineering applications of AI and expert systems (IEA/AIE'98)*, volume 1, pages 584–595.

Jonker, C. M. and Treur, J. (1998b). Agent concepts and agent behaviour: an introduction. Part of the course handouts of the SIKS course on interactive and multi-agent systems, 30 november-4 december 1998.

Jonker, C. M. and Treur, J. (1998c). Information brokering agents applied in a multi-agent architecture for intelligent websites. Part of the course handouts of the SIKS course on interactive and multi-agent systems, 30 november-4 december 1998.

Jönsson, A. (1993). *Dialogue management for natural language interfaces.* PhD thesis, Linköping University, department of Computer and Information Science.

Kim, W. (1995). *Modern database systems.* Addison-Wesley.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C. (1994). Change impact identification in object oriented software maintenance. In *Proceedings of the 1994 international conference on software maintenance.*

Meyer, J. (1998). Intelligent agents: thema-intro en enige theoretische achtergronden. Part of the course handouts of the SIKS course on interactive and multi-agent systems, 30 november-4 december 1998.

Milner, A. J. R. G. (1980). *A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92.* Springer Verlag.

Milner, R. (1993). *The polyadic pi-calculus: a tutorial.* Springer Verlag.

Müller, J. P. (1996). *The design of intelligent agents: a layered approach.* Springer Verlag.

Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., and Doane, P. (1997). The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365.

Nagao, K. and Takeuchi, A. (1994). Speech dialogue with facial displays: Multimodal human-computer conversation. In *Proceedings of ACL-94: the 32nd Annual Meeting of the Association for Computational Linguistics.*

Nielsen, J. (1993). *Usability engineering.* Academic Press, New York.

Pachet, F. (1995). On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4):19–24.

Petrie, C. J. (1996). Agent-based engineering, the web, and intelligence. *IEEE Expert*, december 1996.

Philips (1998). *SpeechMania 2.0: Dialogue Description Language, developer's guide, overhead sheets.* Philips.

Richard, N. and Codognet, P. (1998). Multi-way constraints for describing high-level object behaviours in VRML. In *Proceedings of the Interaction Agents workshop at the AVI'98 conference.*

Richard, N., Codognet, P., and Grumbach, A. (1998). Using constraints to describe high-level behaviours in virtual worlds. In *Proceedings of Eurographics'98.*

Riel, A. J. (1996). *Object-oriented design heuristics.* Addison-Wesley.

Ryan, M. D. and Sharkey, P. M. (1998). Distortion in distributed virtual environments. In *VW'98: Proceedings of the first international Virtual Worlds conference.*

Smith, R. W. (1997). An evaluation of strategies for selective utterance verification for spoken natural language dialog. In *Fifth Conference on Applied Natural Language Processing*, pages 41–48.

Smith, S., Bennett, K., and Boldyreff, C. (1995). Is maintenance ready for evolution? In *Proceedings of the 1995 interational conference on software maintenance.*

Sparck Jones, K. and Galliers, J. R. (1996). *Evaluation natural language processing systems, an analysis and review.* Springer Verlag.

Stenning, K. and Gurr, C. (1996). Formal methods and human communication. In Roast, C. R. and Siddiqi, J. I., editors, *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface.*

van Eijk, R., de Boer, F., van der Hoek, W., and Meyer, J.-J. (1999). Operational semantics for agent communication languages. In van der Hoek, W., Meyer, J.-J., and Witteveen, C., editors, *ESSLLI99 workshop: Foundations and applications of collective agent based systems (CABS).*

van Schooten, B., Donk, O., and Zwiers, J. (1999). Modelling interaction in virtual environments using process algebra. In *TWLT15: Interactions in Virtual Worlds.*

van Schooten, B. W. (1999). Building a framework for developing interaction models: Overview of current research on dialogue and interactive systems. Technical Report TR-CTIT-99-04, University of Twente, Centre for Telematics and Information Technology.

Veer, G. v. d., Mast, C. v. d., Jonker, C., Braspenning, P. J., Wiesman, F., Meyer, J., Poutre, L., and Weigand, H. (1998). Miscellaneous sheets from the 1998 siks course on interactive systems and multi-agent systems. Part of the course handouts of the SIKS course on interactive and multi-agent systems, 30 november-4 december 1998.

Walker, M. A., Litman, D. J., Kamm, C. A., and Abella, A. (1997). PARADISE: a framework for evaluating spoken dialogue agents. *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics.*

Winograd, T. (1972). *Understanding Natural Language.* Academic Press, New York.

Wooldridge, M. (1998). Verifiable semantics for agent communication languages. In Demazeau, Y., editor, *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS 98).* IEEE Press.

Wooldridge, M. (1999). *Intelligent Agents*, chapter 1. MIT Press.

Wooldridge, M., Jennings, N. R., and Kinny, D. (1998). A methodology for agent-oriented analysis and design. In *Agents '99: Proceedings of the Third International Conference on Autonomous Agents.*