Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
Programming Technology Lab

Exception Handling in Ambient-Oriented Systems

Dissertation submitted in partial fulfilment of the requirements for the degree of Licentiaat in de Toegepaste Informatica

Andoni Lombide

Academic Year 2005-2006

Promotor:	Prof. Dr. Theo D'Hondt
Supervisor:	Stijn Mostinckx

May 2006

Abstract

Mobile devices often support connecting to wireless networks and discovering other devices in that network, which provide or require services that involve interaction between the devices. Applications that are conceived as spontaneous interactions between such mobile devices are said to be Ambient Intelligent. The hardware characteristics of such mobile networks have strong repercussion on software development. To cope with the added complexity imposed by these repercussions, the Ambient Oriented programming paradigm has been introduced. This paradigm structures the application components as concurrently executing objects that communicate asynchronously. Keeping the high failure rate and the high uncoupling stemming from the wireless connections connecting the application components in mind, exception handling considerably improves fault tolerance and abstraction. However, in such an Ambient Oriented programming language, classic exception handling such as found in most contemporary object-oriented languages is not enough to deal with failures encountered on application components living on different machines without undermining the autonomy of the mobile devices. This led to the identification of a number of criteria to which an Ambient Oriented exception handling system must adhere. However, these criteria do not impose any encapsulation of the exceptions raised during the interaction of a number of application components, possibly leading to a chain reaction where a large number of application components are being lead to an exceptional state because of their interactions with the components that caused the exception.

Co-ordinated atomic actions are a well-known technique for strongly reducing the complexity of providing fault tolerance in a distributed and concurrent environment by structuring and encapsulating such interactions and integrating them with exception handling such that exceptions are confined in the action and are propagated in a controlled way. Using co-ordinated atomic actions, interactions of application components exhibit transaction-like semantics such that the effects of the interaction are only witnessed when the action can complete, or not at all. This dissertation proposes a slight adaption to the co-ordinated atomic action mechanism to allow it to function in an Ambient Oriented environment. We show how it solves the problem of spreading erroneous information among loosely-coupled application components by using our implementation in a case study.

Samenvatting

Mobiele toestellen kunnen dikwijls via draadloze netwerken andere toestellen in het netwerk ontdekken. Deze toestellen kunnen diensten aanbieden of verwachten die bestaan uit interacties tussen de verschillende toestellen. Applicaties die opgevat zijn als spontane interacties tussen zulke mobiele toestellen worden Ambient Intelligent genoemd. De hardware-eigenschappen van zulke mobiele netwerken hebben een grote invloed op het ontwikkelen van Ambient Intelligent software. Om het hoofd te kunnen bieden aan de extra complexiteit veroorzaakt door die hardware-invloed, werd het Ambient Oriented programmeerparadigma geïntroduceerd. Dit paradigma structureert de applicatiecomponenten als concurrent werkende objecten die asynchroon communiceren. Met de hoge mate van falingen en de hoge graad van onafhankelijkheid van de applicatiecomponenten in gedachten, is exception handling een techniek die veel bijdraagt aan de foutentolerantie en abstractie van een applicatie. Maar in een Ambient Oriented programmeertaal is een klassiek exception handling-systeem niet genoeg om fouten die voorvallen op andere machines af te handelen zonder de autonomie van de mobiele toestellen te ondermijnen. Dit heeft geleid tot de identificatie van een aantal criteria aan welke een Ambient Oriented exception handling-systeem moet voldoen. Deze criteria leggen echter wel geen enkele encapsulatie op van de exceptions die tijdens de interactie van een aantal applicatiecomponenten worden gesignalleerd. Dit kan leiden tot een kettingreactie waarbij een groot aantal applicatiecomponenten in een exceptionele staat wordt gebracht door hun interactie met componenten die een exception veroorzaakten.
Co-ordinated Atomic Actions zijn een gekende techniek om de complexiteit te verminderen van het garanderen van foutentolerantie in een gedistribueerde en concurrente omgeving door het structureren en encapsuleren van interacties van applicatiecomponenten en deze interacties te integreren met exception handling. Met dit model worden exceptions ingesloten in de actie en worden ze gepropageerd op een gecontroleerd manier. Door co-ordinated atomic actions te gebruiken, vertonen interacties tussen applicatiecomponenten een transactionele semantiek zodat de effecten van de interactie alleen zichtbaar worden wanneer de actie gelukt is, en indien ze niet gelukt is, blijven de effecten onzichtbaar. Deze verhandeling stelt een licht aangepaste versie van het co-ordinated atomic action-mechanisme voor om het te kunnen toepassen in een Ambient Oriented omgeving. We laten zien hoe het probleem van zich verspreidende foute informatie tussen zwak gekoppelde applicatiecomponenten kan opgelost worden met Co-ordinated Atomic Actions door onze implementatie te gebruiken in een case study.

Acknowledgements

I would like to thank Prof. Dr. Theo D'Hondt for promoting this dissertation. Special thanks goes to my supervisor Stijn Mostinckx for his advice, his help whenever I encountered problems during this research and for proof-reading this dissertation which improved its quality considerably.
I would also like to thank all the members and students of the Programming Technology Lab for their advice and opinions on the subject, which led to many discussions that contributed to my understanding of the matter. Thanks also to the Vrije Universiteit Brussel for providing such a high standard in the education and teaching of computer science.
Finally, many thanks to my parents for giving me this opportunity and to my friends for their support and distraction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past decade, mobile technology has become increasingly accessible and ever more potent. The mobile devices carried around today include sheer inlimited storage devices (e.g. iPod media players), cell-phones providing worldwide access to the internet, PDAs with ever increasing processing power, etc. Some of these devices can already communicate over short-ranged wireless protocols such as bluetooth. In this thesis dissertation we extrapolate on the hardware that exists today and target interactions between different mobile devices and devices embedded in their surroundings. This vision has been researched for a few years and is known as Ambient Intelligence [?].

Realising this vision requires more than innovations regarding hardware support. At the same time software support is required to deal with networks of mobile devices connected through communication channels which have a limited range. The contribution presented in this thesis dissertation is the construction of an abstraction technique specifically geared towards supporting building software for mobile networks. This work relies on a recent body of work regarding the development of such software and extends it with a new abstraction technique (geared towards exception handling) for grouping concurrently processing devices collaborating with one another in a meaningful way.

1.1 Key Concepts in the Dissertation

The introduced abstraction technique is geared towards handling exceptions in mobile networks and relies on a series of key concepts which are briefly introduced here and further elaborated in their own dedicated chapter.

1.1.1 Ambient Intelligence

Mobile devices often support connecting to wireless networks and discovering other devices in that network, which provide or require services that involve interaction between the devices. Applications that are conceived as spontaneous interactions between such mobile devices are said to be ambient intelligent. Ambient intelligent applications should deal with the following characteristics of mobile networks in an appropriate way: connections are volatile (because of the limitations of wireless technology), resources have to be dynamically discovered (because devices may move in or out of range at any time) and devices should be autonomous and execute their tasks concurrently. These characteristics require a programming language that adheres to the ambient oriented programming paradigm. The ambient oriented programming paradigm dictates that the object model of the language should be classless (to allow software migration) and devices should communicate using non-blocking communication primitives (to preserve device autonomy). Furthermore, the communication traces should be reified in the language (to allow for explicit synchronisation between devices) and a mechanism is needed to discover communication partners (ambient acquaintance management).

1.1.2 Exception Handling

Because of the increased complexity inherent to distributed and concurrent applications, exception handling is a field of ambient intelligent application development that requires careful investigation. A classic exception handling mechanism as can be found in most contemporary object oriented languages cannot be aligned with ambient oriented applications. The reason is that classic exception handling mechanisms do not incorporate support for the concurrent, loosely-coupled and spontaneous interactions of mobile devices over volatile connections into their model. Techniques exist to allow exception handling in a concurrent and distributed environment, but they do not respect all of the ambient oriented properties. Concretely, an ambient oriented exception handling system should allow exceptions to be propagated between different asynchronously executing processes which in turn must be able to handle them in the right context. Furthermore, it should be possible to aggregate concurrently raised exceptions such that they can be collectively handled or concerted to a single exception. When processes collaborate, they should be informed when one of their collaboration partners signals an exception such that the exception can be collaboratively handled by all the participants of the collaboration. Finally, an ambient oriented exception handling mechanism should be loosely-coupled such that the autonomy of the devices is preserved (i.e. it should not rely on a centralised node and should be able to discriminate between long-lasting and temporary disconnections).

1.1.3 Co-ordinated Atomic Actions

As a result of the ambient oriented exception handling requirements mentioned above, exceptions may be propagated from one device to another, including to devices that were not directly involved in the collaboration where the exception originated. This makes it hard to reason about such systems since a whole constellation of devices can be put in an exceptional state by a single failure. This calls for the need of higher level exception handling constructs that allow structuring and encapsulating the collaboration of a number of devices such that the effects of the collaboration are not witnessed by devices outside of the collaboration until the collaboration is over (or signalled failure). One of these constructs is the Co-ordinated Atomic Actions (CAAs) model. CAAs allow structuring collaborations among concurrent processes in a recursive way such that they exhibit transaction-like semantics. No data may breach the CAAs boundaries and if the CAA cannot finish an attempt is made to rollback the participants into the state they were before entering the CAA. Furthermore, exceptions raised by one participant are raised in the other participants too to allow collaborative exception handling. Additionally, the CAA may provide its own exception handlers (for example to rollback the execution).

1.2 Roadmap to the Dissertation

Chapter 2 introduces the ambient intelligence concept along with a scenario exemplifying a series of meaningful spontaneous interactions between different mobile interconnected devices. This chapter further details the hardware characteristics of ambient-oriented software as well as their repercussions on software development process. Finally some common approaches to deal with these repercussions are presented. Chapter 3 provides an overview of classic exception handling techniques found in most object-oriented programming languages and shows why they are not applicable in an ambient oriented system. Furthermore, some existing higher level techniques which are specifically designed to work in a concurrent and distributed system are evaluated from an ambient oriented perspective. Chapter 4 first gives the criteria required for a programming language to be labelled an ambient oriented language. This chapter also presents the existing ambient oriented language AmbientTalk and shows how it deals with the ambient oriented criteria with particular attention for its exception handling model. Chapter 5 shows the results of experimenting with the ambient oriented exception handling constructs provided by AmbientTalk by presenting an implementation of a high level exception handling construct, namely Co-ordinated Atomic Actions. The applicability of the construct is demonstrated by means of a case study. Finally, chapter 6 concludes this dissertation.

Chapter 2

Ambient Intelligence

The recent technological evolutions in combining mobile devices (cell phones, PDA's, ultra-portables...) with wireless technology (WiFi, Bluetooth...) allow mobile devices to break out of their isolation and engage in co-operation with other devices in dynamically changing wireless networks. As devices move about, new networks, services and devices are dynamically discovered. The scale and availability of those mobile networks brings a new breed of co-operative applications, similar to the idea of *ubiquitous computing* [**?**], where both mobile and embedded devices are available everywhere and are seamlessly integrated in the background of our every day life where they may not even be noticed. A single (personal) device is no longer the focal point of users attention and computation is effectively distributed to the environment surrounding the user. To achieve this, devices need to be aware of their location and their surroundings, including other devices, in short: they need to exhibit *context awareness* [**?**]. This vision has been named *Ambient Intelligence* by the European Council's IST Advisory Group [**?**].
The context of pervasive computing implies that the wireless network environment is ad-hoc. This means that there is no assurance on what services are accessible/present in the physical environment. Hence, in an ad-hoc network environment we cannot rely on central server infrastructure [**?**]. This gives rise to new desirable programming constructs not available in most contemporary programming languages. Limited communication range of wireless technology and dynamically appearing and disappearing devices in the network are taken into account in the *Ambient Oriented* programming paradigm. Languages following this paradigm are called *Ambient Oriented* languages [**?**].

In the remainder of this chapter, the hardware characteristics of Ambient Intelligent applications and some typical scenarios are given. Some strategies to deal with the repercussions of these hardware characteristics on software development are described and evaluated.

2.1 Hardware characteristics

The following hardware characteristics of mobile networks increase significantly the difficulty of writing distributed co-operative applications when these issues are not addressed by the underlying programming model.

Connection Volatility Because of the limitations of wireless technology and the fact that devices may move out of range of each other, disconnection and reconnection may happen at any time in the mobile network. Because of this connection volatility, collaborating devices cannot rely on stable network connections and should allow to resume meaningful tasks when a connection is re-established or when a replacement service is discovered.

Ambient Resources Because of the dynamic nature of mobile networks, devices have no explicit knowledge about available resources and the availability of resources may depend on the location of the device. Hence, a mechanism is needed to discover ambient resources as new devices offering services dynamically join and leave the network.

Autonomous Devices Every device should be able to act as an autonomous computing unit and should be able to recover when one of its communication partners disconnects, such that the device does not remain blocked until that communication partner returns. Devices should be able to collaborate without requiring to be connected to infrastructure (servers) to either discover other participants or to co-ordinate the interaction. The underlying network layer should rely only on peer-to-peer connections.

Natural Concurrency To support autonomy, every device should have its own thread and collaborate with other devices in a concurrent fashion. Single threaded applications would freeze an entire network of devices if the device that holds the running thread would disconnect. Communication should happen asynchronously, as explicitly waiting for the result of a request undermines the autonomy of the device. Furthermore, asynchronous communication allows more advanced execution schemes such as redundantly request the same service to different devices in order to ensure better performance, reliability or quality of service. It is necessary to orchestrate the collaborations between naturally concurrent mobile devices.

2.2 Scenario: The Ambient Intelligent travel agency

At the online AmI travel agency, travellers can book a trip to a city of their choice. Subsequently they give some of their favourite pastimes, for example art expo-

sitions, theatre, sports or gastronomical events. However, no specific pastime is planned, they choose concrete pastimes at the beginning of every day itself. When they feel like doing something other than the preferences they had given out earlier, that is possible too.
We follow Luc who just decided to book a city trip to Brussels. As soon as he filled out all the forms on the website and the money transfer has been validated, all the necessary certification and authentication authorities that he can use during his visit are uploaded to his PDA.

When Luc arrives in Brussels, he immediately receives the necessary information - including maps, lists of ongoing events, etc. - on his PDA from the tourist office of Brussels. Given this information, Luc has to decide what events he is going to participate in and negotiate their price. For instance the price of guided tour depends how many people are going to participate. As Luc is a beer enthusiast, he selects a guided visit to some local breweries on his PDA. Since some slots are still available, Luc is allowed to journal a reservation. He receives a price quote, but since there may be more participants joining the guided visit, the price may drop. He thinks the current price is too high and decides not to make a reservation just yet, but he tells the tourist service that he wants to be instantly notified if the price drops.

In the meantime, Luc decides to go to the hotel to drop off his luggage and have lunch. Upon paying for the trip, his PDA automatically received a code that he can currently use to open his hotel room. The code will expire at the end of the trip. Having a quick lunch in the hotel, Luc is all but pleased with the quality of the food at the hotel restaurant, and enters a negative comment on his PDA. His comment will be uploaded to the customer review section of the travel agency, as soon as his PDA is able to connect to the internet. Luc subsequently specifies that he only wants to have his breakfast in the hotel, and that he will go out to lunch for the rest of the trip. This automatically reduces the price he has to pay for his stay at the hotel. Furthermore, restaurants in the neighbourhood that match the traveller's food preferences will send information to his PDA with a form to make a reservation. Our traveller immediately makes a reservation for tonight's dinner in a restaurant that has received a lot of positive reviews from its visitors.

Luc is notified by a message on his PDA that the brewery visit he was interested in has dropped in price. However, if he does not hurry, he will not make it in time. Knowing his location and the starting point of the activity, the PDA consults the city guide and suggests that Luc should take a taxi, or he will be too late. Luc agrees, and after a few minutes he receives a message from a taxi driver in the neighbourhood that he is willing to take him to the meeting point for the guided tour for a certain price. Luc accepts the offer, and walks outside the hotel looking for the taxi.

As the participants join the guide, the guide's PDA detects their presence. When all participants are there, the group takes off to the first brewery.

2.2.1 Evaluation

Here we will use the scenario given above to illustrate the hardware characteristics of an ambient intelligent setting and their repercussions on the software describing the ambient oriented applications.

Connection Volatility As Luc moves around with his PDA, no stable connection can be assumed. Still, Luc should be allowed to do meaningful work even if a required network connection is not available. Upon Luc's arrival, Luc's PDA should download all the necessary city information (maps, event lists...) whenever a suitable connection is available. Later, Luc enters a negative review on his PDA concerning the food of the hotel, which will be uploaded to the customer review section of the travel agency. The application should be able to postpone this process until a connection to the internet is available.

Ambient Resources When Luc is looking for a taxi to reach his destination in time, his PDA automatically discovers a taxi driver in communication range. Both devices have no explicit knowledge about their network addresses, but are spontaneously discovered. Discovery should happen using high-level information, because low-level information - such as network addresses - may not be available or may change frequently in a mobile setting. In a similar fashion, Luc's PDA is aware of the other participants when he arrives at the meeting point for the event.

Autonomous Devices Luc's PDA is able to discover services offered by other devices without assuming any infrastructure.

Natural Concurrency Multiple travellers can concurrently reserve a slot for an event. The application should manage this concurrency to avoid that multiple travellers are assigned in the same slot.

The situations presented in this scenario also illustrate the inherent collaboration among different mobile devices in ambient intelligent applications. Luc's PDA collaborates with the tourist office service to negotiate a price for an event in which Luc is willing to participate. The hotel applications are involved in a collaboration with the customer PDA's. Given the situation, alternative actions may be required, for example because Luc is not pleased with the hotel's restaurant, his PDA looks in collaboration with the devices present at the hotel for replacement services, in this case restaurants in the neighbourhood.

2.3 Dealing with mobile network characteristics

To deal with mobile network hardware phenomena, *time* and *space uncoupling* is needed. Space uncoupling means that the different agents in the mobile network know each other through high-level information (such as names) instead of low-level information (such as explicit hardware addresses). Time uncoupling means that communication between devices is not necessarily the same as transmitting data over the network. Space uncoupling allows programs to move from one device to another and time uncoupling allows devices to do meaningful work even when connections are broken and to resume operations that required a connection that has been re-established. This section explains the two most often used techniques to accomplish these goals, namely tuple spaces and actors.

2.3.1 Tuple spaces

Tuple spaces provide a means of communication among concurrent processes using shared data. A tuple space is a virtual shared memory of elementary data structures - the tuples. Different sets of such tuples may reside on different processors, but to the user the tuple space looks like one single global memory shared between all processes. Tuples are the fundamental data structure of a tuple space and are represented by lists of fields. Tuples are accessed by specifying its contents, as will be exemplified in the next section. Tuple spaces provide interprocess communication and synchronization which are logically independent of the underlying computer or network.
Since tuples are anonymous chunks of data, their retrieval is based on pattern matching on the tuple contents. A template is a possibly incomplete tuple whose fields may contain ordinary values and logic variables. Such logic variables are preceded by a question mark and act like placeholders for data to be retrieved. They are matched against ordinary values when selecting a tuple from the tuple space. The matching algorithm goes as follows: the matched tuple and the template are checked to have the same arity. Ordinary values in the tuple must have the same type, length and value as the corresponding values in the template. A value in the tuple matches with a logic variable in the template if it has the same type and length.

Linda

Linda is a parallel programming library for conventional programming languages that uses tuple spaces as a means of communication among concurrent processes to allow a uncoupled style of computing. Linda's shared tuple space is persistent, globally accessible and statically created. Linda supports parallelism with a small number of simple operations on a tuple space to create and coordinate parallel processes
In Linda, each tuple is a list of typed parameters, that contain the actual information being communicated, and can be accessed concurrently by several processes. Tuples are represented by a list of up to 16 fields, separated by commas and enclosed in parentheses, e.g.: `("arraydata", 13, 2)`.

out(*t*) generates a data (passive) tuple. Each field is evaluated and the resulting tuple is put into the tuple space. Control is then returned to the invoking program. From that point on, the tuple *t* is available for any subsequent operation on the tuple space. The update of the tuple space is performed atomically, which means that no operation can access the tuple space while it is being updated.
For example: `out ("arraydata", dim1, dim2).`

eval(*t*) generates a process (active) tuple. Control is immediately returned to the invoking program. Logically, each field of *t* is evaluated concurrently by a separate process and then *t* is placed into the tuple space.
For example: `eval ("test", i, f(i)).`

in(*p*) uses a *template p* to retrieve a tuple from the tuple space. If multiple tuples match the template, the one returned by **in** is selected non-deterministically and without being subject to any fairness constraint. If no matching tuple is found, the process that executes the operation will *block* until a matching tuple is available. This provides synchronisation between processes. Once a matching tuple is retrieved, it is taken out of the tuple space and is no longer available for other retrievals.
For example both `in ("arraydata", ?dim1, ?dim2, ?dim3)` and `in ("arraydata", 4, ?dim2, ?dim3)` match with `("arraydata", 4, 6, 6+2).`

rd(*p*) proceeds identically to **in**(*p*), except for the fact that a copy of the tuple that matched the template *p* is delivered to the process that executes the operation instead of withdrawing the matching tuple from the tuple space. Similarly to the **in** operation, **rd** is blocking.

Communication in Linda is uncoupled in time and space. Uncoupling in time refers to the fact that senders and receivers do not need to be able to communicate directly with one another in order to exchange information. Tuples are stored in the tuple space and can be retrieved later, even if the process that produced the tuple has terminated its execution already. Uncoupling in space refers to the fact that a tuple in the tuple space is available to the processes dispersed on the nodes of a distributed system - the actual location of the tuple producer and consumer are irrelevant.

Maintaining a tuple space persistent, globally accessible and statically created as Linda assumes, is unfeasible in a mobile environment because connectivity can no longer be taken for granted. Fault tolerance techniques addressing this issue have been proposed [?] [?], but these replication-based techniques were developed under the assumption that disconnection is an exception, whereas in an Ambient Intelligence context, disconnection is rather the rule. Furthermore, these techniques rely on the locality of processes, which is not applicable in an Ambient Intelligent setting as processes may move from one device to another and suspended tasks may be resumed by different processes on different devices.
The following middleware solutions are extensions to the Linda model which attempt to deal with these issues and support the use of tuple spaces as a means of communication in a mobile environment.

LIME: Linda in a Mobile Environment

LIME is middleware which attempts to relieve the programmer of being concerned with the low-level mechanics of communication between mobile hosts and agents. LIME extends the Linda model to function in a mobile environment by transparently altering the set of accessible tuples for a particular host and all agents that reside on it in response to changes in the network topology (of mobile hosts) [?]. The LIME model supports mobile agents (programs) which can travel among mobile hosts (devices) through *transiently shared tuple spaces*, which enable dynamic reconfiguration of their contents according to agent migration or connectivity variations. LIME relies on the concept of *Transparent Context Maintenance*, which supports the shift from a fixed and reliable network to a dynamically changing one by breaking up the monolithical Linda tuple space into various tuple spaces, each permanently associated to a mobile host or agent, and by introducing rules for transient sharing of the individual tuple spaces based on connectivity [?].

The tuple space that is permanently and exclusively attached to a mobile unit (either it is a mobile agent moving in logical space, or a mobile host roaming the physical space) is called its *interface tuple space*. The ITS contains the tuples that its unit has made available to other units. ITS's are at all times co-located with their mobile units and are transiently shared with the mobile units that are currently in range. When a new mobile unit comes into range, tuples in the ITS of the new unit are merged with the ones shared between the the other mobile units. Such an *engagement* is performed as a single atomic operation. *Disengagement* of interface tuple spaces belonging to mobile units moving out of range, results in the atomic removal of data perceived by the remaining units through their ITS's. By default an agent may query tuples regardless of whether it resides in its local tuple space, or in the so-called *federated tuple space*, the merged tuple space of all connected agents. However, Linda operations can be supplied with tuple location parameters to be able to explicitly address the tuple spaces representing different agents. This leads to the problem of misplaced tuples. When a mobile agent wanting to a insert a tuple in another agent's tuple space explicitly by using a location parameter, there may be no connection between the two agents. In this case, LIME places the tuple in the ITS of the agent exporting the tuple, and ensures that whenever the connection becomes available, the misplaced tuple is transferred from the ITS to the correct agent. Because of this, the location parameters consist of the current location of the tuple and the destination location, as a tuple may be moved around until the right destination is found.

LIME also extends the basic Linda tuple space with the notion of a *reaction*. *Strong reactions* of the form **reactsTo**(*s*,*p*), where *s* is a code fragment in the host language and *p* a tuple template, execute a code fragment whenever a tuple matching the given template is inserted in the tuple space. The tuple that matched the template and triggered the reaction is not removed from the tuple space, because multiple reactions may be registered on the arrival of that tuple. In a highly dynamic environment, the ability to respond to changes in the environment is often crucial. However, since reactions are not executed by one particular agent, but rather by the LIME middleware, some additional restrictions apply. Blocking operations are not allowed in the reaction code, as they would block the processing of all the reactions. The only statement that can trigger a reaction is the insertion of a tuple in the tuple space. The same location parameters that were added to the basic Linda operations can be used with reactions, but the current location parameter is confined single host or agent. The reason for this restriction is that the content of the federated tuple space depends on the content of the tuple spaces belonging to physically distributed, remote agents. Thus, to maintain atomicity and serialisation of reactions, a distributed transaction encompassing several hosts for each tuple space operation on any ITS must be started, which is unfeasible in a mobile environment where disconnection happens frequently. LIME also provides *weak reactions*, of the form **upon**(*s*,*p*), which accept no location parameters and work on the whole federated tuple space. The execution of the reaction code of a weak reaction does not happen atomically when a tuple is detected that matches the template of the reaction. The reason is that this would require starting a distributed transaction and suspending the execution of all the connected agents, since the changes made by the weak reaction are applied to the federated tuple space. These changes must be invisible to all connected agents until the reaction code is finished executing if atomicity is required. Instead of taking place immediately in an atomic fashion, weak reactions are guaranteed to take place eventually after such condition, if connectivity is preserved.

These restrictions leave it up to the programmer to evaluate whether his application will cause problems concerning atomicity and serialisation. For example, if an **out**(*t*) operation is executed in the code of a reaction, *t* may match a template specified by some other reaction and thus generate a potentially infinite reaction loop. Furthermore, in a fully mobile setting, disconnection can take place at any time; moving tuples from one ITS to another (e.g. because a tuple is misplaced) may lead to distributed consistency problems and proper application level protocols are needed to prevent tuple duplication or loss.

TOTA: Tuples On The Air

TOTA is a middleware infrastructure combining uncoupled communication with adaptation to contextual information. TOTA uses tuples as a unified mechanism to deal with both context representation and uncoupled interactions among distributed agents. Unlike in LIME, tuples are not attached to a specific node (or to a specific data space) of the network. Instead, tuples are injected in the network and can autonomously diffuse in the network accordingly to a specified rule. These tuples can represent both information to be exchanged between agents as well as more general contextual information on the distributed environment.

TOTA is composed of a peer-to-peer network of possibly mobile nodes, each running a local version of the TOTA middleware. Each TOTA node holds references to a limited set of neighbouring nodes. The structure of the network, as determined by the neighbourhood relations, is automatically maintained and updated by the nodes to reflect dynamic changes, due to mobility or failures. Each node is capable of locally storing tuples and letting them diffuse through the network. TOTA tuples are defined in terms of their contents and their propagation rule. They are injected in the system from a particular node, and spread by hopping to neighbouring nodes according to their propagation rule [**?**]. The contents of a tuple are identical to Linda tuples. The propagation rule determines how the tuple should be distributed and propagated in the network; the distance it should travel before suspending propagation, how it is affected by other tuples it encounters on the nodes it visits, etc. TOTA tuples are not merely distributed replicas: the propagation rule can determine how the tuple's contents change while it is propagated. TOTA constantly monitors the network topology and the arrival of new tuples. The middleware automatically propagates tuples as soon as appropriate conditions occur. For example when new nodes enter the network, TOTA automatically checks the propagation rules of the already stored tuples and it may propagate the tuples to the new nodes.

From the application components' point of view, executing requests and interacting with other agents basically reduces to injecting tuples, perceiving local tuples, and acting accordingly to some application-specific policy [**?**]. TOTA nodes have full access to the local tuple space, and can execute any Linda-like operation on the local tuple space. In addition, TOTA uses a similar mechanism to LIME reactions, such that components can be notified of locally occurring changes to the tuple space (e.g. arrival of certain tuples). In TOTA all queries are to be represented as tuples, such that other components can react to the arrival of such tuple and inject a reply tuple propagating towards the enquiring node.

Evaluation

Tuple space-based approaches provide a very high degree of time and space uncoupling, but this high degree of uncoupling comes at a price. Structuring cooperative activities between different agents in the network becomes much more complex using tuple-based communication. For example, in much cases the order in which tuples are processed is undefined unless the application programmer explicitly introduces a numbering scheme and waits until all necessary tuples are available, causing a possibly infinite wait unless timeouts are used. Moreover, a tuple that may be interesting to an agent may be deleted by another agent (or tuple in TOTA). In addition, if the application requires that some tuples are only to be read or used by a defined set of agents, the programmer is responsible of coming up with a mechanism that ensures that these tuples cannot be touched by other agents, which requires impractical solutions such as sending authentication tuples back and forth over the network. In short, tuple spaces provide no mechanism for abstraction and information hiding and as a result no data protection against arbitrary and improper operations.

2.3.2 Concurrent and Distributed Object Oriented Programming

By separating the specification from the implementation, object-oriented programming provides the modularity necessary for programming in the large. Objects can be defined as entities which encapsulate data and operations into a single computational unit. Object-orientation can be unified with concurrency, and such object-oriented models of concurrent computation must specify how the processes are conceived and how they interact with objects. Object oriented programming builds on the concepts of objects by supporting patterns of reuse and modularity, and concurrency abstracts away some of the details in an execution (namely the ordering of instructions). Concurrency allows in a higher degree of uncoupling of application components [**?**], because concurrent components do not wait for replies unlike their sequential counterparts. Thus it seems natural to bring objects and concurrency together in this distributed context.

There are different ways in which the object paradigm can be used in concurrent and distributed contexts, as described in [**?**]. The *library* approach *applies* object-oriented concepts, such as encapsulation and abstraction, and possibly also class and inheritance mechanisms, to structure concurrent and distributed systems through class libraries in a given object oriented methodology and a given object-oriented programming language. Various components, such as processes, synchronisation means, and name servers, are represented by various object classes (services) with clear interfaces, increasing modularity. In this approach, programming remains mostly sequential object oriented programming, because the library is extended rather than the language. The library approach helps in structuring concurrency and distribution concepts and mechanisms, but keeps them disjoint from the objects structuring the application programs. The programmer still faces at least two different major issues: programming with objects and managing concurrency and distribution of the program, also with objects but not the same objects. For example, Java provides concurrency through the `Java.util.concurrent` package, which provides a number of classes from which objects can be created that represent concurrent components. There is no clear distinction between ”regular” objects and objects dealing with concurrency (such as monitors).
The *integrative* approach consists in unifying concurrent and distributed system concepts with object oriented ones. For example, merging the notions of *process* and *object* gives rise to the notion of an *active object* or *actor*, which will be explained further (the actor model can also be realised using other approaches, see [**?**] for a reflective one). Merging the notions of *transaction* and *object invocation* gives rise to the notion of *atomic invocation*. Integrating distribution can be

done by considering objects as units of distribution. Objects are seen as entities that may be distributed and replicated on several processors. The message passing metaphor is seen as a transparent way of invoking either local or remote objects. Implementing inheritance in this case becomes problematic, as remote code for superclasses may become inaccessible, unless all class code is replicated to all processors, which is unfeasible in an Ambient Intelligent context. A solution is to replace the inheritance mechanism between classes by the concept of *delegation* between objects, as described in [?] (see chapter **??** for an example). Intuitively, an object that may not understand a message will then delegate it to another object called its parent object. The parent will process the message in place of the initial receiver, or it can also delegate it itself further to its own designated parent.
The *reflective* approach integrates protocol libraries within an object based programming language. The idea is to separate the application program from the various aspects of its implementation and computation contexts (models of computation, communication, distribution, etc.), which are described in terms of *metaprograms*.
It should be noted that the library, integrative and reflective approaches are not in conflict but are complementary. For example, the library and reflective approach can both be used in an integrative implementation that supports metaprogramming to implement higher level constructs.

2.3.3 The Actor Model

The actor model [?] can be used as a framework for concurrent object oriented systems. A common approach to modelling objects is to view the behaviour of objects as functions of incoming messages. This is the approach taken in the actor model. Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. This means that the sender of the message does not wait for a response but resumes computation. In the base actor model, primitives are available to create actors from a given behaviour, to send a message to an actor, and to replace the behaviour of an actor with a new behaviour. Sending a message causes the message to be put in an actor's mailbox (message queue). The execution of a message may cause changes to the behaviour of the actor.
When an actor sends a message (passing any number of values) to another actor, it is transparent to the sender whether the receiver is on the same device or a different device [?]. The message must eventually be delivered after a finite but arbitrary long delay, and when it is delivered, it is put in a mailbox. Guarantee of delivery in a mobile setting means concretely that messages which are meant to be delivered to a disconnected device, should be delivered whenever the device is rediscovered in the network. The order in which subsequently sent messages

are delivered is not specified [?]. The receiver executes sequentially the messages in its mailbox in the order they arrive (such that no race conditions can occur). Results of executed messages must be explicitly supplied to the customer actors using asynchronous messages.

The basic actor model provides only a set of low level primitives to build concurrent systems. Higher-level constructs are necessary both for raising the granularity of description and for encapsulating faults. Examples are synchronisation constructs, transactions, error recovery, etc. *Futures* are an example of such a high level construct which diminish the complexity of returning results using asynchronous message passing. A message send immediately returns a promise for a future reply, without waiting for the actual completion of the invocation. Only when the caller really needs the result (i.e. it is used in a primitive operation such as an arithmetic operation or a print operation) is synchronisation with the service provider required. Integration of futures into the invocation mechanism has the effect that the strict synchronisation inherent to synchronous invocation is replaced with synchronisation by need, also sometimes called wait by need.

Most concurrent object oriented languages that use the actor model represent an actor or active object as a sequential process which responds to messages sent to that object. Every active object may execute its actions concurrently. Actors provide a more concise model of concurrency because their operations are encapsulated. There is no interleaving of operations inside a single actor, which means that actors yield sequential semantics internally. This is in contrast with regular concurrent programming where all the operations of processes are interleaved and race conditions must be considered. The actor approach reduces significantly the complexity of concurrent programming and clearly promotes modularity, which can be enhanced by not permitting any shared variable between actors; instead, communication is the primitive by which actors may affect each other [?].
Because actor messages are not anonymous, actors need explicit knowledge about their communication partners. Whereas this avoids the issues mentioned in **??**, it implies that in an Ambient Intelligence context a separate mechanism is needed to discover other actors (possibly residing on different devices) in the network. A solution exists, and will be showed in chapter **??** when explaining an Ambient Oriented programming language using actors.

2.4 Conclusion

In this chapter we described the Ambient Intelligence concept and its repercussions on software development. To cope with these repercussions, two major approaches exist, namely the shared data approach using tuple space-based middleware and the message passing approach staying close to the well-known object-oriented paradigm. Tuple spaces provide a very high degree of time and space uncoupling, but provide no mechanism for abstraction and information hiding and as a result no data protection against arbitrary and improper operations (such as deleting tuples useful to other agents), leaving this burden on the programmer's shoulders. Asynchronous message passing among objects, as used in the actor model, also provides adequate uncoupling, and in addition provides the data encapsulation associated with object orientation. In addition, the actor model also provides a very concise model of concurrency where race conditions do not have to be considered.

Chapter 3

Exception handling

Exception handling mechanisms are the de facto standard to deal with the occurrence of exceptional events in a program. Without exception handling facilities, the programmer is required to explicitly test input as well as return values for exceptional values and change the control flow of the program accordingly; leading to less readable and maintainable programs where the checking for exceptions and recovery is mixed with the regular application logic. In Ambient Intelligent systems, where both distribution and concurrency are implicit, exception handling is a welcome language feature, given the many things that can go wrong in a collaboration of mobile devices (disconnections, failures, data inconsistency...). This chapter further investigates exception handling techniques for passive object systems (section **??**) since most active exception handling systems (section **??**) generalise an underlying primitive passive exception handling system. Finally, these techniques are evaluated from an Ambient Intelligence perspective.

3.1 Exception handling language capabilities

In this section the required capabilities of an exception handling system - as identified in [**?**] - are discussed.

3.1.1 Exception handling for signalling failures

Exceptions are mostly used to model erroneous circumstances. They always indicate one of the following situations that should not be witnessed during a typical execution of a program.

Range failures

Range failure occurs when an operation either finds it is unable to satisfy its output assertions (i.e. its criterion for determining when it has produced a valid result), or decides it may not ever be able to satisfy its output output assertion. To deal with range failures the following capabilities are needed:

- The invoker needs the ability to abort the operation - termination is required. Sometimes as a side-effect of terminating the operation it is necessary to undo all effects of attempting the operation. In a message passing system for example, all messages scheduled for sending but not yet transmitted may need to be retracted.
- The ability to restart the operation, since this may be a reasonable response in some circumstances. For example, a distributed service may fail. It may be possible to try again the same operation using an alternative service.
- The ability to terminate the operation, returning partial results to the invoker, perhaps together with additional information needed to make sense of the results. In our example of the distributed service, the not properly working service may propagate an exception to its customer with the location of an alternative service encoded in the exception.

Domain failures

Domain failure occurs when the input data of an operation fails to pass certain tests of acceptability. To deal with domain failures, the handler must be given enough information about the failure so he can modify the data or supply replacement data to satisfy the unsatisfied criterion. If the handler is unable to fix the problem he must be permitted to terminate the operation, with or without undoing actions taken before the exception was raised. The capability a handler may require to deal with this type of failure is the ability to access the information provided from within the operation.

3.1.2 Exception handling for abstraction

Exception handling mechanisms are not needed just to report errors. They are needed, in general, as a means of conveniently interleaving actions belonging to different levels of abstraction. For example, an addition overflow may be signalled, but as long as the bits of the result are interpreted appropriately by the handler, the operation may resume. This is a cleaner way than using status variables or return codes. Another example where resuming is required, is when using exceptions for monitoring purposes. An operation may raise exceptions to notify the invoker of certain conditions that are not necessarily failure related.

3.2 Passive exception handling

We name an exception handling mechanism passive when it is only concerned with local exceptional conditions in a sequential program, such as the **try-catch** mechanism provided by many contemporary programming languages as a basic language feature. We make the distinction with active exception handling, because we will show in section **??** that in an Ambient Intelligent system passive exception handling is not enough to signal and recover from exceptional situations. Passive exception handling systems allow capturing and redirecting the control flow of a program to a handler that will aid in diagnosing and resolving the exceptional situation. The handler is determined either by static scope rules or by a dynamic invocation chain. Most languages adopt the latter approach, because it increases reusability since the invoker of an operation can possibly handle it in a context-dependent manner. This usually involves traversing the runtime stack from the point where the exception was thrown until a stack frame designating a handler or guarded program unit is found. This process is called exception *propagation*.

3.2.1 Handler semantics

When a matching handler is found for the raised exception, different semantics can be opted for.

Termination semantics

Control flow transfers from the point where the exception was thrown to a handler, terminating intervening executions. Hence, the handler acts as an alternative operation for the operation it protects (cfr. Java, C++, Ada...). In languages that exclusively use this model, the stack frames between the raise and the handler/-guarded unit can be discarded. This is called *stack unwinding*. The termination model where handlers return a value to the invoker is sometimes referred to as the *replacement model*.

Resumption semantics

Control flow transfers from the location where the exception was thrown to a handler to correct the exceptional condition and then back to the throw point, so there is no stack unwinding. This is similar to a routine call, with the difference

that the handler catching the exception is located dynamically.
A problem when employing resumption is *recursive resuming*, meaning that an exception raised in the handler body may be caught again by handlers that are conceptually 'higher' up the execution stack when resuming, creating an infinite loop. However, in [**?**] it is shown that this problem is easily solved by marking all visited handlers when propagating, so they cannot be used again, preventing recursive resuming.
Resumption semantics are very useful in any application in which recoverable exceptions are raised, like interactive applications where users interact to recover from an exceptional situation.
Note that a resuming handler may determine that control cannot return to the throw point and unwind the stack (i.e. terminate the guarded invocation). One mechanism to allow this capability is a language primitive that can be called in the handler to trigger stack unwinding.

Retry semantics

In the retry model, after a handler is executed, the programming unit it guards is restarted. Retry semantics can be simulated by using a loop in combination with termination semantics.

3.2.2 Handling level

Program units of different granularity can be guarded by exception handlers. In an object oriented language, a single expression, a block, an object or a class may be guarded. Expression and block level handling offer more fine grained control and are therefore widely used. With object or class level handling, an object or class is associated with a set of handlers. On the object level, each class instance has a different set, while handling on the class level implies that each instance has the same set as specified in the class. Whenever an exception is thrown inside the object, it is caught if there is a matching handler associated with the object. The advantage is that on the object level, an object handles uniformly all exceptions, and on the class, each instance handles all exceptions uniformly. However, because of the fine grained nature of block level handling, it is more flexible (for instance, different methods of the same object may handle the same exception in a different way) and can mimic object level handling while increasing the expressiveness substantially.

3.2.3 Exception hierarchy

An exception hierarchy is useful to organise exceptions similar to a class hierarchy in class-based object oriented languages. An exception can be derived from another exception, just like deriving subclass from a class. Handlers for a parent type then catch both a derived and parent exception. A programmer can then choose to handle an exception at different degrees of specificity along the hierarchy. Typically, the more specific handlers are given first, which will catch some subtypes of an exception. They are then complemented by more general ones catching less specific exceptions. This allows future extensions of the exception hierarchy. Hence, an exception hierarchy supports a more flexible programming style.

3.2.4 Bound exceptions

Objects are the main components in an object oriented software design, and their actions determine program behaviour. Hence, an exception situation is (usually) the result of an object's action, suggesting the object responsible for throwing an exception may need to be taken into account while catching it [**?**]. For example, a `FileException` may be thrown by different objects, each requiring different corrective behaviour from a different handler. In one case it may be appropriate to create a new empty file, while in another the user may be presented with a dialogue asking him to select the right file. When matching based solely on the exception's type, there is no way to distinguish which object threw the exception. With bound exceptions, it is possible to specify a handler that only catches exceptions thrown by a certain object (the exception is said to be *bound* to that object) and apply corrective behaviour specific to that object.

When propagating the exception, it may be propagated outside the bound object's scope. This makes it intuitive to not only dynamically bind the exception to the handler, but also to the object it propagates into, which actually becomes responsible for the exception, as the propagating object could not handle it. This will shift the binding from an object that does not catch an exception to the object that invoked the operation on the object not catching the exception. This semantics is not easy to encode manually if the language does not support it directly, since it requires constantly updating the responsible object while searching the runtime stack for an appropriate handler for the thrown exception. For an implementation example see section **??**.

3.2.5 Exceptions as First Class Objects

Representing exceptional events as classes and each of its concrete occurrences as an exception instance in an object oriented language holds numerous advantages [?]:

- Exception hierarchies are automatically mapped onto class hierarchies. Different classes denote different categories of exceptions.
- Class variables allow sharing information between different exception instances.
- Signallers can communicate with handlers by passing to handlers the instance of the signalled exception which holds in its instance variables all the information about the exceptional situation.
- New user defined exceptions can be created as subclasses of existing ones and are uniformly signalled and handled.

3.2.6 Disciplined exception handling

The fundamental principle behind disciplined exception handling is that a routine must either succeed or fail; either it fulfils its contract or it does not. In the latter case an exception is always raised. This approach is used in the Eiffel language [?] [?]. Contracts can be violated in several ways, all of which are considered *faults* and represented by exceptions. Operating environment problems, such as running out of memory, are one situation in which exceptions are signalled. In these cases a contract can fail, but not necessarily because the caller or the callee did something wrong.
In Eiffel, contract violations fall into two categories. In the first category, preconditions of a method are not met, for which the caller is held responsible. In this case, an exception is signalled in the caller. The second category is where method post-conditions or loop invariants are not met. In this case, the callee is held responsible and an exception is signalled locally.
A handler is attached to a method as a rescue clause, which can re-execute the method using a retry statement after it has restored the object to a consistent state and possibly patched things up. Alternatively, the rescue clause can also act as an alternative body for the method. If handling the exception is not possible or results in a failure, the rescue clause in its turn signals a failure and propagates an exception.

3.3 Ambient Oriented Exception Handling

Exception handling in ambient oriented systems yields new challenges, such as dealing with asynchronous communication, connection volatility, software migration, etc. Classic exception handling systems are not designed to deal with these phenomena and fall short in providing adequate support. New language abstractions for exception handling are needed to ease the development of applications which are based on spontaneous interactions between different mobile devices.

3.3.1 Ambient Oriented Exception Handling Criteria

Asynchronous Exception Propagation

When an exception is thrown, a handler is searched for in a *dynamically defined context*. Typically, such a context is defined by means of a **try-catch** construct, where the **try** block describes the context for which the handlers may handle exceptions.
In an ambient oriented setting, the **try-catch** construct is not applicable due to the use of *non-blocking communication primitives* for the reasons discussed in **??**. As a result, the calling process may have left the context of its **try** block before the exception was propagated by the invoked process. Therefore, an ambient oriented exception handling system should provide an adequate mechanism for ensuring that exceptions raised by concurrent processes are caught in the correct context [**?**].

Concerted Exceptions

The combination of non-blocking communication primitives with block level handling implies that all processes invoked by the block may concurrently raise exceptions, as the executions of the asynchronous messages happen concurrently. These exceptions may or may not be related, but in many cases should be handled jointly. An ambient oriented exception handling mechanism should therefore allow the programmer to examine all concurrently raised exceptions and to subsequently propagate a *concerted* exception which best captures the particular exceptional situation.

Collaborative Exception Handling

Ambient oriented programs are conceived as a collaboration of processes which should be able to continue working in the face of volatile connections. Therefore, the individual processes typically make optimistic assumptions while performing their tasks. Once an exception raised by one process violates these assumptions, not only the user of the computation, but also the other processes involved need to be informed of the exception so as to enable a co-ordinated recovery.

Loosely Coupled Exception Handling

An ambient oriented exception handling system should guarantee the autonomy of the processes for which it handles exceptions. This implies that it may not rely on a centralised node to co-ordinate exception handling. Furthermore, it needs to provide a mechanism to discover long-lasting disconnections, in order to prevent processes from waiting indefinitely for an unreachable communication partner.

3.3.2 Message level handling

As discussed earlier, two approaches exist to support non-blocking process communication: the asynchronous message passing approach and the shared data approach, represented using tuples in a shared distributed tuple space. This is the finest level of granularity at which a distributed exception handling mechanism can operate.

Asynchronous Messages

The occurrence of an exception can be propagated among processes that communicate by message passing as a special call-back message. In ABCL/1 - a concurrent object oriented language based on the active object concept - a message has both a reply destination and a complaint destination [**?**]. The reply destination indicates the object to which the receiver should send his reply message. If the receiving object encounters an exception it cannot handle itself during the execution of the message, it sends an exception message to the complaint address included in the message. The object designated by the complaint address may in its turn handle the exception or propagate it to another object.
As discussed earlier, futures or promises provide a way of handling results and exceptions produced by asynchronous invocations. The E language uses futures to provide synchronisation-by-need when using asynchronous messages sends. Asynchronous invocations in E return a placeholder object (the future) which will be dynamically *resolved* to the real return value of the invocation whenever that becomes available. When the execution of a message results in an uncaught exception, the future will resolve to the raised exception that will be reraised in the caller. The promise is said to be broken, and messages sent to that promise are said to be contaminated. This is called *broken promise contagion* [**?**].
To enable the use of results and the handling of exceptions resulting from asynchronous invocations, E provides the **when-catch** expression. A **when-catch** ex-

pression takes a future, a **when** block to execute if the future resolves to a value, and a **catch** block to execute if the promise is broken. By using a **when-catch** construct, a process can postpone the execution of a block of code until it has gathered enough information from another process. This synchronisation-by-need between caller and callee implies that exceptions are automatically propagated into the correct scope. Consider the example of a stack that executes requests asynchronously and returns a future for each request. When one of its clients requests popping the stack, it immediately returns a future that may be resolved to an exception - instead of the top of the stack - if the stack is empty. With the **when-catch** construct, it is possible to observe that future until it is resolved to either the top of the stack or an `EmptyStackException`, and then execute the appropriate code. The client may have been doing other work while waiting for the future to be resolved, but when it resolves to an exception, the exception will still be handled in the context were the client needed the top of stack. Client and stack are synchronised-by-need when the future is resolved.

Tuple Spaces

When independent agents communicate by exchanging tuples in a tuple space, exception handling mechanisms protect tuples emitted by agents. When representing exceptions as ordinary tuples, one cannot rely on the eventual handling of the exception should the tuple never be read. Therefore, the CAMA system proposed in [**?**] is based on LIME and introduces an additional mechanism for redirecting exceptions to a remote host and handling it by a different agent or by a special handler code left by the agent before migrating. Redirection means implicitly sending and re-raising the exception in a different location. The exception may pass through several locations before it reaches the agent, which makes it necessary to employ some mechanism to preclude loops and excessively long travel paths. Handling delegation can be used if there is a friendly agent that can perform exception handling when the original agent is not present in the location. Such a friendly agent may be just a spawned version of the original agent that handles exceptions or a dedicated stationary agent that handles exceptions for a whole class of mobile agents. CAMA knows the originator of each tuple to prevent tuples representing exceptions to be read accidentally.
To be able to find a handler by routing exceptions to the handling agent, agent-specific information is needed to compute that route. CAMA therefore requires every tuple to be equipped with a reference to a special tuple called a *tuple space trap* to which exceptions are signalled. Such a tuple space trap has enough information to be able to transform the exception and find a route to propagate it to the ”caller”, a dedicated handler agent or an ensemble of (affected) agents. Agents interested in handling tuple space exceptions can produce these tuples. Tuple space

traps can be updated or removed at any time thus enabling dynamic exception handling patterns.

To be able to build such a system and remaining reliable and predictable, the federated tuple spaces used in LIME are replaced by an approach that is based on a stationary and persistent tuple space. This is a major drawback, because it implicitly assumes infrastructure. An assumption which is not supported by the ambient oriented model.

Evaluation

Message level approaches can be used as a basis to develop an ambient oriented exception handling system, but they lack support for funnelling concurrently raised exceptions to a single *concerted exception*, since they consider only one message send and thus one exception at a time.

3.3.3 Block level handling

Various distributed exception handling mechanisms offer a variation to the well-known **try-catch** construct, to bind a single exception handler to a sequence of instructions possibly containing asynchronous message sends. Because of the asynchronous message passing, different exceptions may be raised concurrently in one block. Different mechanisms exist to reduce these concurrent exceptions to a single *concerted exception*.

ProActive

ProActive [**?**] is a Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. ProActive makes use of active objects communicating through asynchronous message sends, which return futures. ProActive attempts to make asynchrony and concurrency transparent by extending the Java exception handling system with support for futures. Therefore, concurrently raised exceptions need to be resolved automatically to a single exception which will be handled by the specified catch block. As a consequence, ProActive only handles the first exception to be raised inside the **try** block, assuming that all asynchronous invocations are closely related such that the first exception is a good representative of the exceptional situation. ProActive reimplements an exception stack, side by side with Java's original. Unfortunately, because of lack of reification of the call stack in Java, the programmer is supposed to manually ensure consistency between the two stacks by calling dedicated ProActive methods.

SaGE

SaGE [?] is an exception handling mechanism for multi-agent systems that maps a list of concurrently raised exceptions to a single concerted exception using a *resolution function*. Resolution of concurrent exceptions is based on the use of a concert method, which is to be implemented by a service which is co-ordinating a collaboration of independent agents. The concert method is invoked whenever an exception is raised. Based on a log containing previously thrown exceptions, the method may opt to ignore the exception, add it to the log, or propagate an exception (possibly a newly created one) to its caller.

Arche

Arche [?] is a distributed and concurrent object oriented programming language where each object has its own process. Arche allows grouping such active objects into aggregates which permit the parallel invocation of a so-called multi-operation. Although objects normally communicate synchronously, the introduction of multi-operations may lead to concurrently raised exceptions by different aggregated objects. In Arche, each object remains blocked until its call returns or the exception which is raised during the call is handled. When using a multi-operation, a concerted exception is computed from the concurrently raised exceptions by means of a resolution function specified by the programmer. The computation of a concerted exception may not be defined implicitly because it requires semantic knowledge about exceptions [?]. However, when no resolution function is supplied, a default concerted exception is raised. A default concerted exception is computed as follows:

- When at least two different exceptions are raised concurrently, a default failure exception is raised.
- When all concurrently raised exceptions are the same, this exception is raised.

The difference with the SaGE approach is that by blocking until all calls have returned, the concerted exceptions cannot be raised prematurely, thus giving the resolution function information which is guaranteed to be complete.

DOOCE

DOOCE (Distributed Object Oriented Computing Environment) [?] is an extension to C++ that aims to integrate distribution transparently in an object oriented

environment. To achieve this, DOOCE relies on the active object and asynchronous message passing paradigm and uses futures to provide synchronisation-by-need.
DOOCE introduces two language constructs to manage concurrency. First of all, the **par** construct delimits a block of statements wherein messages are sent concurrently. After replies of all these messages are received, the code next to the **par** block is executed. Additionally, messages can be annotated with an **async** qualifier. If the message passing statement qualified by an **async** qualifier is executed, it returns a future and the code next to it is executed immediately without waiting for its actual reply. If the caller accesses the future while the requested result has not been received yet, caller and callee are synchronised by blocking the caller until the result is available.
DOOCE provides programmers with the flexibility of either handling each concurrently raised exception in a block individually or grouping all exceptions together and handling them as a single exception. To this end, when exceptions are raised concurrently, each exception may be handled independently by a **catch** block, such that multiple of these blocks may be executed as each one catches one of the raised exceptions.
DOOCE extends the C++ **try-catch** construct with **catch** statements that take multiple exceptions as argument. The **catch** block will only be executed if all exceptions have been raised concurrently during the execution of the **try** block it guards. Therefore, **catch** statements that catch multiple exceptions should be defined earlier than the **catch** statements that take a single exception, else those may consume certain combinations of exceptions, which as a result would never be caught by the **catch** blocks tailored for the exceptional situation that the raised exceptions collectively describe.

Evaluation

The DOOCE approach is clearly more flexible as it does not force the programmer to map a number of concurrent exceptions onto a single concerted one. For example, different **catch** blocks may be used for case analysis, rethrowing the caught exception. However, resolution functions do provide a intuitive way of mapping concurrently raised exceptions onto a single concerted exception, but this can be implemented as a higher level construct based on a handler that catches multiple exceptions.
Despite their provisions for producing concerted exceptions, these mechanisms do not qualify as an ambient oriented exception handling mechanism, since they require the concerted exception to be handled solely by the sender of the message, offering no collaborative exception handling. This implies that the techniques described above are only applicable when the different processes make no optimistic

assumptions, which are often required in an ambient oriented setting to deal with volatile connections.

3.3.4 Collaboration level handling

Some libraries or middleware allow structuring an application as a complex interplay of different processes. In addition to the mechanisms they provide for structuring such interactions, they also provide mechanisms for handling exceptions that may be raised concurrently by those processes.

Open Multi-threaded Transactions

OMT transactions [?] structure a group of collaborating threads within the boundaries of a transaction. The OMT model allows threads to join an ongoing transaction at any time and to be forked and to terminate inside a transaction. There are only two rules that restrict thread behaviour: a thread created outside a transaction cannot terminate inside the transaction, and a thread created inside a transaction must also terminate inside the transaction. The threads participating in the same transaction communicate using shared objects (called *transactional objects*) that maintain their own consistency, using conventional techniques such as mutual exclusion.
OMT transactions can be nested. A participant of an OMT transaction can start a new (nested) transaction. Sibling transactions populated by different participants execute concurrently. That is to say, a thread can only participate in one sibling transaction at a time. To join a nested transaction, a thread must be a participant of the parent transaction. Any participant of a transaction can decide to *close* it at any time. Once the transaction is closed, no new participants can join the transaction. Accesses to transactional objects by participants are isolated from accesses by other transactions, even when the transactions are in a parent-child relation.
All transaction participants *finish* their work inside the transaction by voting on the transaction outcome. Possible votes are *commit* and *abort*. The transaction commits if and only if all participants commit. In this case, the changes made to transactional objects on behalf of the transaction are made visible to the outside world. If any of the participants wishes to abort, the transaction aborts. In that case, all changes made to transactional objects on behalf of the transaction are undone. Joined participants are not allowed to leave the transaction, i.e. they are blocked until the outcome of the transaction has been determined. This means that all joined participants of a committing transaction exit synchronously. If a participating thread ”disappears” from a transaction without voting on its outcome, the transaction is aborted.
When a participating thread raises an exception and does not handle it, it is propa-

gated to the OMT transaction. This exceptions is immediately transformed into a `Transaction_Abort` exception that is raised in all the participants that voted `commit`, and is also propagated to the parent transaction, if any. The OMT transaction itself is aborted. This implies that OMT transactions do not support *collaborative exception handling* among the different participants.

Co-ordinated Atomic Actions

An *atomic action* or *conversation* encloses an interactive activity of a group of processes such that there are no interactions between that group and the rest of the system for the duration of the activity, as such preventing erroneous information from spreading throughout the whole system. By excluding the rest of the system from the interaction and possibly raised exceptions, atomic actions provide a means of encapsulating interactions such that their effects are not witnessed until they succeed or fail. When an exception is raised that cannot be handled locally, it will be propagated to all processes participating in the conversation.
An extension to this model are co-ordinated atomic actions. A co-ordinated atomic action (CAA) is designed as a stylised multi-entry unit with action roles for the different participants co-operating within the CAA. Logically, the action starts synchronously when all action roles have been activated and finishes when all of them reach the action end. The action can be completed either when no error has been detected, or after successful recovery, or when a failure exception has been propagated to the containing action. If an error is detected all participants are involved in recovery [**?**]. When several exceptions are concurrently raised in an action, they are resolved using either resolution trees/graphs or resolution functions (see section **??** for possible approaches), and a resolved exception is propagated to all action participants, which will be jointly responsible for recovering the system co-operatively. This means that interacting processes co-operate not only when they execute the normal program behaviour but also when they recover the program [**?**].
External (transactional) objects can be used concurrently by several CAA's provided that they offer a way to ensure that information cannot be smuggled among them and that any sequence of operations on these objects bracketed by the CAA start and completion has the ACID[1] properties with respect to other sequences. A CAA execution behaves like an atomic transaction to the outside world. A participating process can only leave the interaction when all of the participants have finished their roles and the external objects are in a consistent state. This is needed to guarantee that if something goes wrong in the activity executed by one of the participants, then all participants can try to recover from possible faults [**?**].

[1]Atomicity, Consistency, Isolation and Durability

The CAA concept allows designers to associate exception handling with modularised interactions between different processes.
When a CAA is not able to tolerate an error, a failure exception is propagated to the containing action passing the responsibility for recovery to the higher system level and leaving the objects involved in the action execution in well-defined states, thus facilitating the recovery at the higher level [?]. However, the CAA model requires every participant to exit the CAA with the same result. This implies that once a participant disconnects, the entire collaboration must be aborted, which is too rigid for structuring collaborations in an ambient oriented system. Still, a relaxed adaptation of the CAA abstraction may prove to be useful.

The Guardian Model

The Guardian exception handling model [?] [?] uses a distributed global handler (the *guardian*) to orchestrate the exception handling action by directing each involved process to the correct local handler. The directing is done by raising in each process the appropriate exception, which may differ from the exception raised in another process. The global handler uses application defined *recovery rules* to determine which exception it raises in each participant, which in turn causes the correct exception handler to be invoked.
The Guardian model allows associating a symbolic name to an exception context. The context of a distributed application is the union of all individual participant contexts. The purpose of contexts is to provide a mechanism to invoke correct exception handlers. When an exception is raised in a process, a target context is specified in the exception object, since the raising context and the target context may not be the same. Contexts provide a means to give a dynamic meaning to an exception based on the current program flow, and are used to ensure that there is a handler for an exception or that all processes have a common execution region they can be rolled back to. For each process, a context stack is maintained and each context has an associated list with exceptions it can handle. Participating processes are responsible for pushing and popping the right context on the stack, to allow the correct handlers (defined in the participants' program code) to be invoked should the participant encounter an exception.
At each suspend point (state where the processes may be suspended) of the execution, the guardian checks whether one or more participants encountered an unhandled exception (processes raising an exception are immediately blocked). If this is the case, the guardian blocks all participants at this point to ensure that all concurrently signalled exceptions are known to the guardian. The guardian evaluates the recovery rules given the exceptions signalled, and all context lists. A matching rule defines, for each participant, the target context and exception to raise in that participant. The exceptions defined by the recovery rules are collec-

tively raised in their respective participants. The recovery rules may map a single signalled exception into another kind of derived exception for each participant or alter the context to adequately handle the exception. After the exception is raised locally, exception handlers are searched in the participant using the participant's runtime environment. The guardian compares the exception's target context with the encountered handler's context. If both match, the exception is handled, else another handler is looked for.
To the Guardian, participants are identified using their context lists. An identifier represents a subset of participants whose current context list matches the specified identifier. A fully qualified identifier includes the entire context list as the identifier, while a partially qualified identifier is expressed as a regular expression of context names. Using a partially qualified identifier allows greater flexibility in identifying a subset of participants.
Since each participating process may change its context independently, the Guardian model needs to contact all processes whenever exceptions are raised. To be able to do this, the Guardian model assumes total-order reliable group communication primitives and a timed asynchronous model. This requirement implies that the exception handling mechanism depends on the presence of all processes, which may not be possible in systems relying on loosely coupled exception handling.

Evaluation

OMT transactions try to minimise the effect of a single exception, so that it may always be handled locally. This provides no collaborative exception handling. Co-ordinated atomic actions implement transaction-like guarantees to ensure that the effect of errors can be adequately handled collaboratively by its participants. CAA semantics are too rigid for ambient oriented applications, but a relaxed version of the CAA model may be useful. The Guardian model does not impose the use of a transaction-like system. The Guardian model makes the notion of an exception context explicit, and recovery rules bring participating processes back into a certain explicitly specified context to handle the exception in collaboration with the other participants, who may be in a different context and handle a different exception. This is in contrast with co-ordinated atomic actions, where exception contexts are implicitly created by entering a new CAA, with its associated exception handlers and exception resolution mechanisms. This implies that all exceptions raised within a CAA are automatically handled in the right context. The lack of such an imposed structure makes the Guardian model impractical in an ambient oriented setting. The added flexibility of the Guardian model provided by the ability to raise different concerted exceptions in different participants is not really needed when using a transaction-like construct. For instance, in the CAA model the same concerted exception is raised in each process. This is no problem as the concerted exception reflects well the current exceptional situation described by the concurrently raised exceptions, since they all belong to the same CAA context.

3.4 Conclusion

We have given an overview of current practice in passive and active exception handling and have evaluated the different approaches from an ambient oriented point of view. Ambient oriented exception handling requires:

- Exceptions to be propagated between different asynchronously executing processes which must be able to handle them in the right context.
- Concurrently raised exceptions to be aggregated so they can be collectively handled or concerted to a single exception.
- Different collaborating processes to be informed of exceptions raised in their collaboration partners to enable collaborative exception handling.
- A loosely coupled exception handling mechanism that guarantees the autonomy of the process for which it handles exceptions by not relying on a centralised node and by discovering long-lasting disconnections.

In an asynchronous message passing system, futures seem to be a satisfying synchronisation mechanism which also allows exceptions to be propagated among asynchronous processes. An E-like **when-catch** construct allows futures to be entirely non-blocking and allows capturing the right context for handling both results of asynchronous messages and exceptions.
The DOOCE approach of catching multiple concurrently raised exceptions in a block provides a flexible way of creating a context where exceptions may be raised concurrently and handled collectively or mapped to a single concerted exception that reflects the exceptional situation.
A relaxed version of the co-ordinated atomic action model may prove to be useful as exceptions are isolated in an implicitly created context in which they can be collaboratively handled by the participating processes in the action.

Chapter 4

AmbientTalk

AmbientTalk [?] was conceived as a reflectively extensible ambient oriented language kernel. AmbientTalk's design is directly based on the analysis of the hardware phenomena mentioned in **??** and features a number of fundamental semantic building blocks designed to deal with these hardware phenomena. AmbientTalk is used as a language laboratory that allows investigating which language features are essential to the ambient oriented programming paradigm. In this chapter, the AmbientTalk language is discussed and additional focus is laid upon its passive and active exception handling systems.

4.1 Ambient Oriented Programming

The ambient oriented programming paradigm presents a way of dealing with the hardware characteristics of ambient oriented applications mentioned in section **??** by mapping them onto language features of a concurrent object-oriented language. In this section, the language design characteristics that discriminate the ambient oriented programming paradigm from classic concurrent distributed object-oriented programming languages are presented.

4.1.1 Classless Object Models

When sending a remote message, the objects passed as arguments are copied to the receiving host. Since an object in a class-based programming language cannot exist without its class, the classes of the copied objects have to be copied and sent over the network as well. However, a class is - by definition - an entity that is conceptually shared by all its instances. From a conceptual point of view there is only one single version of the class on the network, containing the shared class variables and method implementations. The copying of classes as a consequence of argument passing, combined with the ability to update class variables and methods yields a classic distributed state consistency problem among replicated classes. To avoid such problems caused by implicit sharing relations at the language level, ambient oriented programming languages make all sharing relationships explicit such that they can be controlled by the programmer. This design decision yields a prototype-based language, where objects are conceptually entirely idiosyncratic such that the above problems do not arise. For these reasons, it has been decided to select prototype-based object models for ambient oriented programming. Notice that this confirms the design of existing distributed programming languages such as Emerald, Obliq, dSelf and E which are all classless.

4.1.2 Non-Blocking Communication Primitives

Recall from section **??** that the autonomy of mobile devices is of paramount importance in the context of Ambient Intelligence. Upholding this autonomy may prove to be very difficult when the devices communicate using synchronous, blocking communication. Not only does blocking communication give rise to potential distributed deadlocks (which are extremely hard to resolve in mobile networks since not all parties are necessarily available for communication), but it also induces a strong dependency from the caller to the callee. The former may remain blocked indefinitely when the volatile connection with its communication partner is broken. Therefore, an ambient oriented programming language should not allow its objects to explicitly wait for (the result of) a call from another object.

4.1.3 Reified Communication Traces

When asynchronously communicating devices are collaborating autonomously, they may end up in an inconsistent state. Devices need to be able to restore their state to a previous consistent state they were in, such that they can synchronise anew based on that final consistent state. To allow this, an ambient oriented programming language has to reify the communication traces that led to the inconsistent state. Having such an explicit reified representation of whatever communication happened while both parties could not synchronize, allows a device to properly recover from an inconsistency by reversing (part of) its computation.

Reified communication traces are also useful to be able to implement different message delivery policies. For example, in the actor model, eventual delivery of messages is guaranteed. In this case, this concretely means that a copy of the sent message should be kept at the sender until it is acknowledged that the message reached its destination. In fact, it may be required to resend the message in case it is lost or to postpone the sending until the receiver is available.

4.1.4 Ambient Acquaintance Management

Mobile devices should not need an explicit reference or network address to each other or other ambient resources beforehand (whether directly or indirectly through a server) to be able to communicate. Instead, they have to dynamically discover each other in the network while roaming the environment. For this discovery, and to interact with each other, mobile devices should not rely on a third party (as opposed to client-server architectures), because this undermines their autonomy. It may be possible to set up a server for the purposes of a particular application. However, an ambient oriented programming language should allow an object to spontaneously get acquainted with a previously unknown object based on an intentional description of that object rather than via a fixed URL. Incorporating such an acquaintance discovery mechanism, along with a mechanism to detect and deal with the loss of acquaintances, should therefore be part of an ambient oriented programming language.

4.2 The AmbientTalk Language

AmbientTalk is conceived as a proof by construction that languages can be conceived which abide the four characteristics described above. This section describes the language, shows its usage and identifies how it fulfills the requirements outlined in the previous section. Finally, the reflective capabilities used in the remainder of this dissertation are detailed.

4.2.1 Object Model

AmbientTalk has a concurrent object model that is based on the ABCL actor model [**?**]. AmbientTalk actors consist of a perpetually running thread, updateable state, methods and a message queue. These concurrently running actors communicate by asynchronous message passing. Upon reception, messages are scheduled in the actor's message queue and are processed one by one by the actor's thread. By excluding simultaneous message processing, race conditions on the updateable state are avoided. This way, AmbientTalk unifies imperative object-oriented programming and concurrent programming using the integrative approach (see **??**) without suffering from omnipresent race conditions.
AmbientTalk's object model is double-layered: it distinguishes between active and passive (i.e. ordinary) objects. This allows programmers to deal with concurrency only when strictly necessary (i.e. when considering semantically concurrent and/or distributed tasks) and avoids having every single object to be equipped with heavyweight concurrency machinery and having every single message to be

thought of as a concurrent one. Since passive objects are not equipped with an execution thread, the current thread runs from the sender into the receiver, thereby implementing synchronous message passing. However, it is important to ensure that a passive object is never shared by two different active ones because this easily leads to race conditions. To avoid this, every passive object is contained within exactly one active object. Therefore, a passive object is never shared by two active ones. The only thread that can enter the passive object is the thread of its active container. In order not to violate this containment principle, passive objects that are passed as arguments to an asynchronous message sent to an active object are always passed by copy. This means that the passive object is deep-copied up to the level of references to active objects. Active objects process messages one by one and can therefore be safely shared by two different active objects. Hence, they are passed by reference.
Active objects are defined to be AmbientTalk's unit of distribution and are the only ones allowed to be referred to across device boundaries. Therefore, AmbientTalk applications are conceived as suites of active objects deployed on autonomous devices. Several active objects can run on a device and every active object contains a graph of passive objects. Objects in this graph can refer to active objects that may reside on any device. In other words, AmbientTalks remote object references are always references to active objects. The rationale of this design is that synchronous messages (as sent to passive objects) cannot be reconciled with the non-blocking communication characteristic presented in **??**.

4.2.2 Passive Object Layer

AmbientTalk passive objects are conceived as collections of slots mapping names to objects and/or methods. The code below shows an implementation for stacks in AmbientTalk:

```
makeStack()::object({
 els: makeVector(10);
 top: 0;
 isEmpty()::{ size=0 };
 isFull()::{ size=top };
 push(item)::{
   if(this().isFull(),
     { error("Stack Overflow") },
     { top:=top+1;
       els.set(top,item) })
 };
 pop()::{
   if(this().isEmpty(),
     { error("Stack Underflow") },
     { val: els.get(top);
```

```
      top:=top-1;
      val })
  }
})
```

AmbientTalk is a prototype-based language, this implies that objects have no associated class [?], but are cloned or extended from an existing prototype. Objects are created using the `object(...)` primitive. It creates an object by executing its argument expression, typically a block of code (delimited by curly braces) containing a number of slot declarations. Objects have a private mutable part (defined with the `:`-operator), a public constant part (defined with the `::`-operator) and a single parent pointer allowing for single inheritance. The constants are shared between cloned objects. When cloning an object, its private variables will be deep-copied and the constants will be shallow-copied. This means that candidates for sharing (typically methods) should be declared constant, whereas object-specific data (i.e. instance variables) should be defined as variables. Both method invocation and public slot selection use the classic dot notation. The function `makeStack()` is referred to as a constructor function and is AmbientTalk's idiom to replace the object instantiation role of classes.
Objects can also be created by extending existing ones. The code below extends the stack object with a push method that automatically increases the stack size to prevent overflow:

```
makeSafeStack()::extend(makeStack(), {
  push(item)::{
    if(this().isFull(),
      { els.makebigger() });
    super().push(item)
  }
})
```

`extend(p, ...)` creates an object whose parent is `p` and whose additional slots are listed in a block of code, analogous to the `object(...)` form. Slot lookup follows the delegation semantics as prescribed by Lieberman [?]. If a slot matching the requested selector is not found in the receiver, lookup is delegated to the parent. The receiver (e.g. the result of `this`) still points to the initial receiver of the message.

Apart from objects, AmbientTalk features built-in numbers, strings, a null value `void` and functions. However, these 'functions' are actually nothing but methods in AmbientTalk. For example, the `makeStack` constructor function is actually a method of the root object which is the global environment of the AmbientTalk interpreter. Methods can be selected from an object (e.g. `myPush: aStack.push`). Upon selection, a first-class closure object is created which encapsulates the receiver (`aStack`) and which can be called using canonical syntax, e.g., `myPush(element)`. Closure objects are actually passive objects with a single apply method. Finally, a syntactic sugar coating allows anonymous closures to be created given a list of formal parameters and a body, e.g.,

```
lambda(x,y) -> { x+y }
```

When bound to a name (e.g., as the value of a slot `f` or when bound to a formal parameter `f`), a closure is called using canonical syntax, e.g., `f(1,2)`.

4.2.3 Active Object Layer

As explained in section **??**, AmbientTalk actors have their own message queues and computational thread which processes incoming messages one by one by executing their corresponding method. Therefore, an actor is entirely single-threaded such that state changes using the classic assignment operator := cannot cause race conditions. Messages sent to the passive objects it contains (using the dot notation) are handled synchronously. Actors are created using the `actor(o)` form where `o` must be a passive object that specifies the behaviour of the actor. In order to respect the containment principle (see **??**), a copy of `o` is made before it is used by the actor form because `o` would otherwise be shared by the creating and the created actor. A newly created actor is immediately sent the `init()` message and `thisActor` denotes the current actor. These concepts are exemplified by the following code excerpt which shows the implementation of a friend finder actor running on a cellular phone. When two friend finders discover one another (which is explained later on) they send each other the match message passing along an info passive object that contains objects representing the age (with an `isInRangeOf` method) and hobbies (containing a method that checks whether two hobby lists have anything in common).

```
makeFriendFinder(age, hobbies)::actor(object({
  init()::{ display("Friend Finder initialized!") };

  beep()::{ display("Match Found - BEEP!") };

  match(info)::{
    if(and(age.isInRangeOf(info.age),
          hobbies.intersectsWith(info.hobbies)),
      { thisActor()#beep() })
  }
}))
```

Actors can be sent asynchronous messages using the # primitive which plays the same role for actors as the dot notation for passive ob jects. E.g., if `ff` is a friend finder (possibly residing on another cellular phone), then `ff#match(myInfo)`

asynchronously sends the `match` message to `ff`. The return value of an asynchronous message is `void` and the sender never waits for an answer. Using the `#` operator without actual argument (e.g., `ff#match`) yields a first-class message object that encapsulates the sending actor (`thisActor`), the destination actor (`ff`) and the name of the message. First-class messages are further explained in section **??** that describes AmbientTalks meta-level facilities. Finally, using the dot notation for actors (resp. `#` for passive objects) is considered to be an error.
When passing along arguments with (both synchronous and asynchronous) message sends, caution is required in order not to breach the containment principle. In the case of synchronous messages of the form `po.m(arg1,...,argn)` between two objects that are contained in the same actor, the arguments do not "leave" the actor and can therefore be safely passed by reference. In the case of asynchronous messages of the form `ao#m(arg1,...,argn)`, the arguments "leave" the actor from which the message is sent. In order to respect the containment principle, this requires the arguments to be passed by copy as explained in section **??**. In the friend finder example, the info object is thus passed by copy.

4.2.4 First Class Mailboxes

AmbientTalks concurrent object model presented above is classless and supports non-blocking communication. This already covers two of the four characteristics of ambient oriented programming as presented in section **??**. However, with respect to the other two, the model presented thus far still has some limitations which it directly inherits from the original actor model:

- The model does not support the ambient acquaintance management characteristic of the ambient oriented programming paradigm because traditionally, actors can only gain acquaintances through other actors.

- Actor formalisms do not support the reified communication traces we argued for in section **??**, because although actors have a queue of messages to be delivered, they provide no reified access to this queue.

To enable these two properties, AmbientTalk replaces the single message queue of the original actor model by a system of eight first-class mailboxes which is described below.

Reifying communication traces

When scrutinising the communication of a typical actor, four types of messages are distinguished: messages that have been sent by the actor (but not yet received by the other party), outgoing messages that have been acknowledged to be received, incoming messages that have been received (but not yet processed) and messages that have been processed. The AmbientTalk interpreter stores each type in a dedicated mailbox associated with the actor. An actor has access to its mailboxes through the names `outbox`, `sentbox`, `inbox` and `rcvbox`. The combined behaviour of the `inbox` and `outbox` mailboxes was already implicitly present in the original actor model in the form of a single message queue. AmbientTalks mailboxes are the fundamental semantic buidling blocks for implementing advanced language constructs on top of the non-blocking communication primitives. Indeed, conceptually, the mailboxes `rcvbox` and `sentbox` allow one to peek in the communication history of an actor. Likewise, the mailboxes `inbox` and `outbox` represent an actors continuation, because they contain the messages the actor will process and deliver in the future. Together, the four explicit mailboxes cover the need for reified communication traces that have been prescribed by the ambient oriented programming paradigm.

Enabling Ambient Acquaintance Management

In order to cover the ambient acquaintance management requirement of ambient oriented programming, AmbientTalk actors have four additional predefined mailboxes called `joinedbox`, `disjoinedbox`, `requiredbox` and `providedbox`. An actor that wants to make itself available for collaboration on the network can broadcast this fact by adding one or more descriptive tags (e.g. strings) in its `providedbox` mailbox (using the `add` operation described below). Conversely, an actor that needs other actors for collaboration can listen for actors broadcasting particular descriptive tags by adding these tags to its `requiredbox` mailbox. If two or more actors join by entering one anothers communication range while having an identical descriptive tag in their mailboxes, the mailbox `joinedbox` of the actor that required the collaboration is updated with a *resolution object* containing the corresponding descriptive tag and a (remote) reference to the actor that *provided* that tag. Conversely, when two previously joined actors move out of communication range, the resolution is moved from the `joinedbox` mailbox to the `disjoinedbox` mailbox. This mechanism allows an actor not only to detect new acquaintances in its ambient, but also to detect when they have disappeared from the ambient.

Mailbox operations

Mailboxes are first-class passive objects contained in the actor. Apart from the eight built-in mailboxes, actors can create their own custom mailboxes which might be used by reflective extensions to temporarily store messages. Mailboxes

provide operators to add and delete elements (such as messages, descriptive tags and resolutions): if b is a mailbox, then b.add(elt) adds an element to b. Similarly, b.delete(elt) deletes an element from a mailbox. Moreover, the changes in a mailbox can be monitored by registering observers with a mailbox: b.uponAdditionDo(f) (resp. b.uponDeletionDo(f)) installs a closure f as a listener that will be triggered whenever an element is added to (resp. deleted from) the mailbox b. The element is passed as an argument to f.
The following code excerpt exemplifies these concepts by extending the friend finder example of the previous section with ambient acquaintance management in order for two friend finders to discover each other. The initialisation code shows that the actor advertises itself as a friend finder and that it requires communication with another friend finder. When two friend finders meet, a resolution is added to their joinedbox, which will trigger the method onFriendFinderFound that was installed as an observer on that mailbox. This resolution contains a tag slot (in this case "<FriendFinder>") and a provider slot corresponding to the providing actor. The latter is sent the match message (as described in the previous section).

```
makeFriendFinder(age, hobbies)::actor(object({
  beep()::{ display("Match Found - BEEP!") };

  match(info)::{
    if(and(age.isInRangeOf(info.age),
           hobbies.intersectsWith(info.hobbies)),
       { thisActor()#beep() })
  };

  onFriendFinderFound(aResolution)::{
    aResolution.provider#match(makeInfo(age, hobbies))
  };

  init()::{
    provided.add("<FriendFinder>");
    required.add("<FriendFinder>");
    joinedbox.uponAdditionDo(this().onFriendFinderFound)
  }
}))
```

4.2.5 AmbientTalk as a Reflective Kernel

The built-in mailboxes and their observers (installed with uponAdditionDo and uponDeletionDo as described above) can already be regarded as part of AmbientTalks meta-object protocol (MOP) since they partially reify the state of the interpreter. Indeed, they constantly reflect the past and future of the commu-

nication state between actors as well as the evolving state of the ambient. Additionally, the MOP (which is only for the concurrent active object model, there is no MOP for passive objects) allows a programmer to override the default message sending and reception mechanisms. This section presents the various operations of the MOP which have a default implementation residing in any actor and can be redefined by overriding them in any idiosyncratic actor.
In order to explain the MOP, it is crucial to understand how asynchronous messages are sent between two actors (that might reside on different machines). When an actor `a1` sends a message of the form `a2#m(...)`, the interpreter of `a1` creates a first-class message object and places it in the `outbox` of `a1`. After having successfully transmitted that message between the interpreter of `a1` and the interpreter of `a2`, the interpreter of `a2` stores it in the `inbox` of `a2`. Upon receiving a notification of reception, the interpreter of `a1` moves the message from the `outbox` of `a1` to the `sentbox` of `a1`. `a2` processes the messages in its `inbox` one by one and stores the processed messages in the `rcvbox` of `a2`. Each stage in this interplay (namely message creation, sending, reception and processing) between the two interpreters is reified in the MOP.

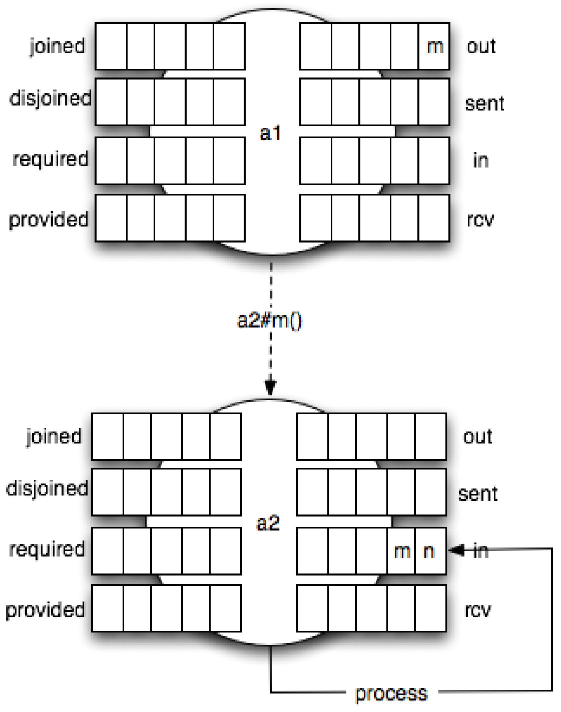

Figure 4.1: AmbientTalk actor communication

Message creation is reified in the MOP with the constructor function `createMessage(sender, dest, name, args)` which generates first-class messages. A message is a passive object which has four slots: the sending actor `sender`, the destination actor `dest`, the name of the message `name` and a vector object `args` containing the actual arguments. Remember from section **??** that a first-class message is also created upon field selection with an expression of the form `anActor#msgName` which results in a first-class message with sender `thisActor`, destination `anActor`, name `msgName` and an empty argument vector.

Message sending is reified in the MOP by adding messages to the `outbox` which is accomplished by the MOP's message sending operation `send`. In other words, an expression of the form `anActor#msg(arg1 , ..., argN)` is base-level terminology for an equivalent call to the meta-level method `send`, passing along a newly created first-class message object. The default behaviour of send is:

```
send(msg)::{ outbox.add(msg) }
```

It is possible to override this behaviour by redefining the method `send`. The example below illustrates how `send` can be overridden for logging purposes.

```
send(msg)::{
  display("sending..."+msg.getName());
  super().send(msg)
}
```

Every actor has a perpetually running thread that receives incoming messages in the `inbox` and transfers them to the `rcvbox` after processing them. **Message reception** is reified in the MOP by adding messages to an actors `inbox` which can be intercepted by adding an observer to that mailbox. **Message processing** is reified in the MOP by invoking the parameterless `process` method on that message (which will execute the recipients method corresponding to the message name) and by subsequently placing that message in the `rcvbox`. The latter event can be trapped by installing an observer on that mailbox.

4.2.6 Conclusion

The AmbientTalk kernel supports the ambient oriented programming paradigm by:

- Employing a prototype-based object model consisting of both passive and active objects. Passive objects are never shared between active objects which are single threaded units of distribution.

- Active objects communicate using non-blocking communication primitives. Asynchronous messages are processed sequentially.

- Communication traces are reified through the use of first-class mailboxes. The MOP consists of these mailboxes, their associated operations and first-class messages and their associated operations.
- AmbientTalk supports ambient acquaintance management to allow actors to discover communication partners in the network.

Using the MOP, the kernel is extended with new language features to ease the development of ambient oriented applications.

4.3 Extending the Kernel : Ambient-Oriented Language Features

In the previous section we have presented an overview of the basic building blocks provided by the AmbientTalk language kernel. Whereas these suffice to make AmbientTalk an ambient-oriented language, they do lack support for common software development concerns. These concerns are catered for by the introduction of new language constructs using the reflective API described in section **??**. In this section we outline techniques to handle return values of actors and an appropriate exception handling mechanism.

4.3.1 Non-blocking Futures

AmbientTalk's implementation of futures (which have been already discussed in section **??** and for their use in exception handling in section **??**) is based on E. In AmbientTalk, futures are non-blocking, meaning that while the operations that require the value of the future are waiting, the actor as a whole is not blocked and may be executing other operations (this corresponds to the need for non-blocking communication primitives as explained in section **??**). Futures are represented as actors and messages sent to them are transparently forwarded. To achieve this, AmbientTalk's default message sending behaviour is reflectively extended for asynchronous message sends to return such non-blocking futures as proposed in the language E [**?**]. The idea is to return a new future whenever a message is asynchronously sent, also if the target is a future actor. When the latter future is eventually resolved to a value, the message is sent to that value and its result will resolve the new future. This is called future pipelining or promise pipelining. Future pipelining allows one to "chain" asynchronous message sends, even though the intermediate results are not yet computed.
Often, synchronisation is needed between the sender and the receiver of an asynchronous message because the application logic dictates that a certain action is to

be undertaken upon resolving a future. Therefore, AmbientTalk and E feature a **when** construct that takes two parameters: a future and a closure, consisting of a formal parameter name for the resolved value and a code block (in which the value the future resolved to can be referred to using the name passed as formal parameter). The idea is that the code block is *registered* with the future. Using the **when** construct itself will not block but return a future in its turn. That future is resolved with the value resulting from executing the code block that was registered with it. The point is to asynchronously schedule this code block for execution *when* the future that corresponds to its first argument has been resolved. As such, **when** allows one to send an asynchronous message resulting in a future (i.e. the first argument) and to specify what should be done upon getting a result (i.e. the second argument), without resorting to blocking and without having to manually establish a connection (e.g. by passing context information) between the time of sending the message and the time of receiving a result. Notice that several uses of **when** can register a code block with the same future. All these blocks will be executed upon resolution of the future.
The **when** construct is exemplified with the following code excerpt which shows how a future resulting from an asynchronous message send can be used to register two different **when** constructs. Executing this code excerpt will immediately display `"first"` on the screen. When the future itself is eventually resolved, `"second"` and `"third"` will be displayed along with the computed result.

```
{ fut: anActor#compute();
  when(fut, becomes(result)
  -> { display("second", result) });
  when(fut, becomes(result)
  -> { display("third", result) });
  display("first")
}
```

4.3.2 Exception Handling

Since the idea behind AmbientTalk's passive exception handling system was to use it as a basis for experimenting with ambient oriented exception handling mechanisms, I have designed a mechanism of which flexibility was the primary goal.

Block Level Handling

To associate a programming unit with one or more handlers is known as protecting or guarding the unit. Depending on the exception handling system, handlers can protect programming units of different granularity, like expressions, routines,

objects...
Similar to most contemporary languages such as Java, C# and Smalltalk, AmbientTalk allows protecting a particular block with a series of handlers (cf. the **try-catch** construct). This mechanism provides a much finer granularity than e.g. object-level handling where an object needs to provide a **handle** message which addresses all exceptions thrown by that object. Furthermore, since AmbientTalk does not feature statements, each individual expression can be protected in the following way :

```
try(expression, handler1, handler2, ..., handlerN)
```

First Class Handlers

Handlers are first-class language values, consisting of the exception they will catch and a body of code that will be executed to remedy the exceptional condition. Handlers may be specified separately from **try** statements, allowing for their reuse. Moreover, handlers are dynamically scoped, meaning that they can access the scope in which they are used, rather than the one in which they were defined. Note that the difference is only relevant when handlers are defined apart from the **try** block in which they are used. This implies that the traditional **try-catch** still performs its expected behaviour, with the handler being able to access local variables defined outside the **try** block. The ability to define dynamically scoped handlers independent of such **try** blocks contributes to the reusability of handlers.

```
catch(exceptionToHandle, handlerCode)
```

When the `throw` native is invoked, the runtime stack is searched for handlers. To make sure that a handler is executed in the dynamic context it catches the exception, the environment is restored (this is not to be confused with stack unwinding as will be explained later in this section) when return frames[1] are encountered while searching the stack. Handlers will always be executed in the context where they caught the exception with an extension to the dictionary that contains the `currentException` variable, which makes it possible to access the thrown exception inside the handler body.

Exception Objects

Exceptions are represented as regular AmbientTalk objects. The basic `exception` object's interface is:

[1]Return frames contain dictionaries that are to be restored when returning from function calls, method invocations...).

exception.new(message) Creates a clone of the exception object.

exception.throw() Throws the exception.

exception.canBeHandled(handlerExc) Checks if the exception can be handled by a handler created from `handlerExc`. This function, which is always invoked by the interpreter when an exception is thrown and a handler is found on the stack, provides a hook to define one's own behaviour to determine whether an exception can be caught. This is achieved by simply overriding the default behaviour in an extended exception.

exception.getType() Returns the type of the object it was originally copied from (using `new()`). Since clones of an object are unconnected in prototype-based languages, the link to the original prototype is maintained by means of a type to allow clones of the same exception to be considered equal in the implementation of the `canBeHandled` method.

exception.getParent() Returns the exception's parent in the hierarchy, this is the exception object it was extended from. This function is used in the default implementation of `canBeHandled`, when the handler's exception type does not match the thrown exception, to test whether the handler catches a parent type of this exception.

For the exact reason why we employ explicit pointers to parent and prototype exceptions, we refer to appendix **??**.

Exception Hierarchy

As mentioned in **??**, an exception hierarchy adds to the flexibility of the system. It is supported by deriving new exceptions from already existing exception objects using the `extend` method. This method takes a code-block (whose evaluation is delayed) prescribing the methods and state of the extension. Child exceptions "inherit" all the data and methods from the exception they are extended from. By default, handlers that are created from a more general exception (an exception higher up in the hierarchy), will handle all its derived exceptions. However, this default behaviour can be changed by overriding the `canBeHandled` method for an extended exception.

```
notFoundException::exception.extend({
  selector: void;
  getSelector()::selector;

  withSelector(sel)::{
    selector:=sel;
    copy()
  };

  canBeHandled(handlerException)::{
    if(and(prototype = handlerException.getPrototype(),
        or(handlerException.getSelector() = void,
           handlerException.getSelector() = selector)),
      { true },
      { parentType.canBeHandled(handlerException) })
  }
})
```

In this example the `canBeHandled` method is overridden to make sure that `notFoundException` handlers catch only exceptions with a matching selector. Note that the call to `parentType.canBeHandled` implies that more general exception handlers (in this case for the exception type) can still be used to handle exceptions of this type.

Bound Exceptions

In AmbientTalk, exceptions are always bound. This means that handlers can access the object *responsible* for throwing an exception by invoking the `boundTo` method on it. Notice that there is a subtle yet important difference between this responsible object and the object that actually throws the exception. Consider the implementation of slot lookup in a prototype-based language, given below. The algorithm traverses the parent chain whenever the slot is not locally available, and when no parents are available, the `notFoundException` specified in the example above is thrown.

```
lookup(aSlotName)::{
  if(slotMap.contains(aSlotName),
    { slotMap.get(aSlotname) },
    { if(slotMap.contains("parent*"),
        { slotMap.get("parent*").lookup(aSlotName) },
        { notFoundException.withSelector(aSlotName).throw() }
    }))
}
```

When protecting a call `o.lookup("non-existing-slot")` with a handler, the object `o` will be considered responsible for the `notFoundException` (and will thus be returned by `boundTo()`), rather than the top-most parent object, who has actually thrown the exception. This semantics is achieved by constantly updating the responsible object while searching the stack for an appropriate handler. Concretely, whenever return frames are encountered on the stack (signalling that a method was called), the responsible object will be updated as shown in figure **??**.

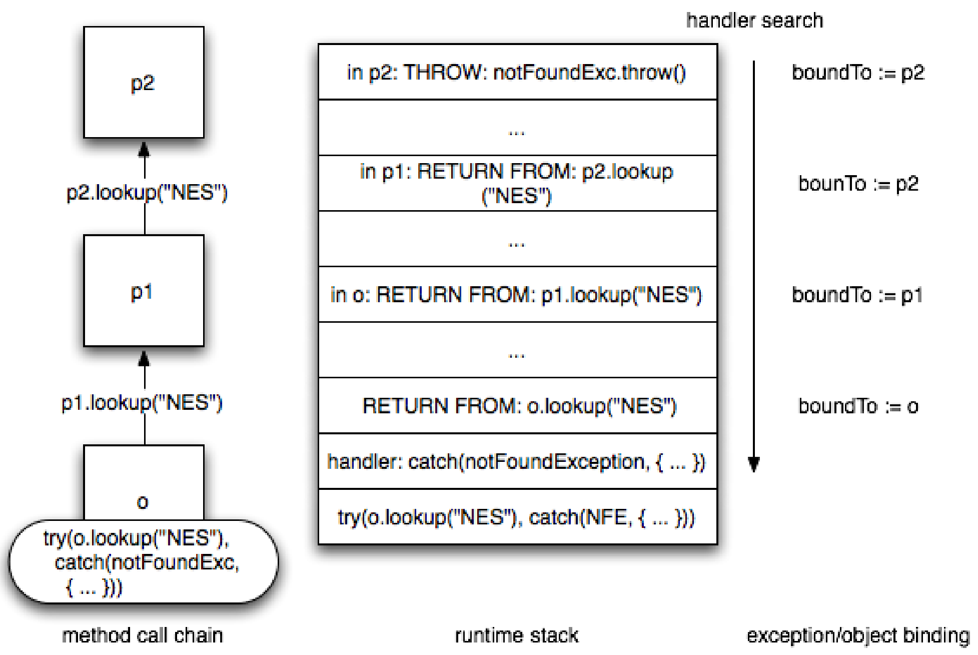
handler search
p2
p2.lookup("NES")
p1
p1.lookup("NES")
o
try(o.lookup("NES"), catch(notFoundExc, { ... }))
in p2: THROW: notFoundExc.throw()
...
in p1: RETURN FROM: p2.lookup ("NES")
...
in o: RETURN FROM: p1.lookup("NES")
...
RETURN FROM: o.lookup("NES")
handler: catch(notFoundException, { ... })
try(o.lookup("NES"), catch(NFE, { ... }))
boundTo := p2
bounTo := p2
boundTo := p1
boundTo := o
method call chain
runtime stack
exception/object binding

Figure 4.2: Updating bound exceptions while returning from method calls

Manually fixed bindings An exception may also be manually bound to a particular object using the `bindTo(obj)` method. This function may be used in two different ways. An exception in a handler may be manually bound to only trap exceptions that originate from a particular object. This technique allows a more fine-grained handling of exceptions, and improves the flexibility of the exception handling mechanism. Furthermore, an exception may also be manually bound before it is thrown. This allows the throwing object to specify the exact location where corrective behaviour needs to be applied.

Handling Semantics

After locating an appropriate handler on the runtime stack and executing it, standard handler semantics resumes the computation after the exception is handled by returning control to the point on the stack where the exception was thrown (similar to returning from a function call). However, an `unwind` native function is available that will unwind the runtime stack to the point were the exception was thrown. Using this native, termination semantics can be obtained, or more complex semantics depending how `unwind` is used.

```
catch(myException, {
  handlerOperation1;
  handlerOperation2;
  ...
  handlerOperationN
})
```

Standard resumption semantics.

```
catch(myException, {
  unwind({
    handlerOperation1;
    handlerOperation2;
    ...
    handlerOperationN
  })
})
```

Termination semantics using `unwind`.

```
catch(myException, {
  handlerOperation1;
  if(nonResumableCondition,
    { unwind({
        terminationOperation1;
        ...
      })
    },
    { resumeOperation1;
      ...
    })
})
```

More complex semantics.

The parameter that is passed to `unwind`, is the return value of the unwind statement. Recall that using termination semantics, the handler acts as a replacement for the faulty code and thus has to return an appropriate result. Important to note is that `unwind` reduces the stack to the position of the **try** block. Therefore any computation that was on the stack (including the remainder of the handler) is lost. This is why the handler code is specified as the return value of the `unwind`.

4.4 Active Exception Handling

For the reasons discussed in section **??**, classic exception handling mechanisms relying on the **try-catch** construct can not be applied in ambient oriented applications. For this reason, AmbientTalk provides an ensemble of four language constructs implemented on top of the passive exception handling mechanism which

directly correspond to the four criteria described in section **??** [**?**].
First of all, the **when-catch** construct allows *asynchronous exception propagation* at the granularity of a single asynchronous invocation. Secondly, exceptions raised in a sequence of several asynchronous invocations may be dealt with using a single exception handler by wrapping this sequence in a **group-resolve** construct. This construct allows treating such a sequence as a single asynchronous invocation, funnelling all concurrently raised exceptions into a single *concerted exception*. Thirdly, *collaborative exception handling* is achieved using the **conversation** language construct. This construct specifies a set of participants and will ensure that exceptions raised by a single participant will be handled by all available participants. Finally, *loosely-coupled exception handling* is introduced through the **due** construct.

4.4.1 Asynchronous Exception Propagation

Asynchronous exception propagation is supported through the use of futures. When an exception is raised inside a method body, the method is said to *propagate* an exception instead of returning a value. The future that is 'waiting' for the result of that method is then said to be *ruined* by the exception. The fact that futures can be ruined by exceptions changes the future pipelining semantics described above in **??**. As explained, when a message `m` is sent to a future `f1`, a new future `f2` is returned that will be resolved by the result of sending `m` to the resolution of `f1`. However, when `f1` is eventually ruined, `f2` will be ruined by the same exception. A similar phenomenon exists in the E language where it is called broken promise contagion [**?**]. In addition to this, exceptions may be propagated when they are the return value of a method, i.e. when a method was asynchronously invoked and responded by sending a message to another actor.
In addition to the future propagation rules outlined above, asynchronously propagated exceptions may also be explicitly handled using an extension of the **when** construct defined in section **??**, namely a **when-catch** construct that allows a programmer to react to ruined futures in an appropriate way. The **when-catch** construct requires three constituents: a future, a block of code to be executed when the future gets resolved to a value, and a set of handler blocks that might be executed when the future gets ruined by an exception. The construct looks as follows:

```
when(fut, becomes(val)
-> { 'when block to execute when fut is resolved' },
catch(Exception1,
  { 'catch block to execute when Exception1 is raised' }),
catch(Exception2,
  { 'catch block to execute when Exception2 is raised' }),
'...'
)
```

Recall that the `->` notation is syntactic sugar for creating a closure (the same approach is used to bind the value of the resolved future to a variable in the **becomes** closure).
The idea is that several **when-catch** constructs can register themselves with a future `fut` and that every **when-catch** construct can list several **catch** clauses. Every k'th **when-catch** registered with a future `fut` denotes a future $\mathbf{f}'_k$ in its turn. If `fut` gets resolved with a value, all registered **when** blocks will be executed. If `fut` is ruined by an exception, all registered **catch** clauses - acting as exception handlers - are notified of the exception, such that they can determine whether one of their branches catches the raised exception (see section **??** where it is explained how it is determined whether a handler will catch an exception or not). When this is the case, the corresponding branch is executed. Both cases (i.e. executing the **when** block or executing the **catch** block) can result in a value being returned or an exception being raised. In the former case, the value is used to resolve $\mathbf{f}'_k$. In the latter case, the exception is used to ruin $\mathbf{f}'_k$.

The **when-catch** construct allows one to postpone certain actions until the result of an asynchronous invocation is known, or resulted in the propagation of an exception. This allows an event-driven style of programming in which the order of execution of some independent code blocks depends on the order in which asynchronous invocations return their results or signal an exception.

4.4.2 Grouping Concurrent Exceptions

By default, the result of a block of code is the value returned by the last expression in that block. Consequently, exceptions propagated from different asynchronous invocations in a block are ignored unless they ruin the future of the last expression (using the propagation rules explained above), which is almost never desirable since exceptions are raised for a reason. For this reason, a mechanism is required to funnel all possible concurrent exceptions and produce a single *concerted exception*.
AmbientTalk provides the **group-resolve** construct as an alternative mechanism to group the exception handling of multiple asynchronous invocations. Unlike an ordinary code block, the **group** clause does not immediately return the value of its last expression. Instead a future is returned, the value of which will be only determined after *all* futures created within the **group** clause either have been resolved with a return value or ruined by an exception. When none of the futures was ruined, the result of the **group-resolve** construct is equivalent to that of an ordinary code block, namely the value of the last expression. However, if exceptions

were propagated, the **resolve** clause will be triggered with an array of concurrently raised exceptions. The **resolve** clause may either return a value (if the reported exceptions can be tolerated or if the exceptional situation can be corrected) - acting as a handler for multiple concurrent exceptions - or raise a *concerted exception*. The construct looks as follows:

```
group({
  'Block containing multiple asynchronous invocations'
}, resolve(concurrentExceptions)
-> { 'Catch all concurrent exceptions'
    'Possibly raise a concerted exception'
})
```

All concurrently raised exceptions in the **group** clause will be grouped in an array that is made accessible to the **resolve** closure through the `concurrentExceptions` variable passed as a formal parameter.
When using a **when-catch** construct inside a **group** clause, exceptions that have ruined a future but were subsequently handled by that nested **when-catch** should not be considered any more by the **group-resolve** construct.

4.4.3 Collaborative Exception Handling

The criteria for an ambient oriented exception handling mechanism mentioned in section **??** stipulate that a mechanism is needed to inform a set of collaborating actors when one of them has propagated an exception. Such an exception might invalidate the optimistic assumptions the actors have to make to achieve a *loosely-coupled exception handling* mechanism.
AmbientTalk provides *collaborative exception handling* through the introduction of a **conversation** abstraction. The conversation's task is to provide a mechanism to propagate exceptions to all participants of the collaboration it embodies. Conversations are represented as actors and are automatically created by the **conversation** construct shown below.

```
conversation(participants, {
  'Body of the conversation'
})
```

When creating a **conversation**, the actors that will participate in the conversation are passed along in an array. The conversation actor itself offers a `propagate` method which, when passed an exception object, broadcasts this exception to all participants such that it can be handled collaboratively. Concretely, the conversation actor checks for each incoming message whether it contains a future (and thus is a reply from a participant which resulted from an asynchronous invocation from within the conversation), and registers a **when-catch** observer on it. When the future eventually resolves to a value, the value is just returned. When the future is ruined by an exception (propagated by the participant that executed the message), the `propagate` method of the conversation is called, which results in raising the exception in each participant. This will trigger the **catch** blocks catching the propagated exception in all the participants, causing collaborative exception handling. To achieve this, each participant has to implement a `startConversation` method, which will be called when the participant enters the conversation. The purpose is to specify the handlers to invoke should the conversation propagate an exception in that method, as shown in the example below.

```
aParticipant: actor(object({
  startConversation(conv)::{
    when(conv, becomes(value)
    -> { 'code to execute when conversation ends succesfully' },
    catch(convException1,
      { 'collaborative handler code for convException1' }),
    catch(convException2,
      { 'collaborative handler code for convException2' }),
    ...)
  };

  'Rest of the actor behaviour...'
}
```

A conversation can thus be thought of as a special kind of future which can be 'ruined multiple times' (`conv` in our example). In addition to providing the `propagate` method, the conversation also has access to the participants, and it can be provided with additional behaviour that is to be specified in the body of the **conversation** construct.

Although a conversation is conceptually thought of a single actor, the *loosely-coupled exception handling* criterion clearly prohibits an ambient oriented exception handling mechanism to introduce dependencies on a single "leader" device (i.e. the device hosting the conversation). Such dependencies are avoided by providing each participant of the conversation with its own local replica of that conversation actor (and thus including all the behaviour specified in its body). Each participant is given a reference to its local replica using the `startConversation` method presented above. A participant can broadcast an exception to all other participants in the conversation by invoking the `propagate` method on its local replica.

4.4.4 Loosely-coupled Exception Handling

An ambient oriented exception handling model must provide for exceptions to be handled in a loosely-coupled fashion. This implies that tight bonds between de-

vices (which harm that device's autonomy) should be avoided whenever possible. It is for this reason that replication was introduced in the design of the **conversation** language construct, to minimize the dependencies between the participants other than those inherent to the task at hand.
Additionally, dependencies between devices are also created when sending messages to actors on another device. Although such dependencies cannot leave a device blocked and unable to respond to requests, the inability to communicate with its communication partner may prohibit the application from making any progress. An extreme example hereof is the **group-resolve** construct which observes a collection of futures and will only end when *all* futures have been either resolved or ruined. In order to make this construct more suitable for an ambient-oriented setting, support is needed to differentiate between temporary (tolerable) and long-lasting (presumably permanent) disconnections. This technique allows loosening the bonds between collaborating actors and allows actors to find replacement services for unavailable collaboration partners.

To address the problem described above, the **when-catch** construct is extended with a **due** clause. The clause allows discriminating temporary from long-lasting disconnections by putting an expiration deadline (in milliseconds) on the resolution (or ruining) of the future observed by the **when** clause, and handling the situation by providing its own handler code. The resulting **when-catch-due** construct can be used as follows:

```
when(aFuture, becomes(result)
-> { 'code to execute when future is resolved' },
catch(Exception1,
  { 'handler code for Exception1' }),
catch(Exception2,
  { 'handler code for Exception2' }),
'... more handlers...',
due(maxTimeOut,
  { 'handler code for expired future' })
```

Notice that `aFuture` can be any statement that returns a future, including for example a **group-resolve** that contains multiple message sends, leading to the expiration deadline to be in effect for each of these message sends.

4.5 Conclusion

In this chapter we have discussed AmbientTalk, an ambient oriented programming language using an integrative implementation of the actor model to allow concurrency and distribution in an object-oriented language. We have shown that

AmbientTalk adheres to the following requirements to be labelled an ambient oriented programming language:

- AmbientTalk employs a classless object model.
- Active objects communicate using non-blocking communication primitives.
- Communication traces are reified through the use of first-class mailboxes.
- AmbientTalk supports ambient acquaintance management to allow actors to discover communication partners in the network.

Since AmbientTalk's basic model attributes no return values to asynchronous messages, it also does not cater for exceptions being propagated from such invocations. Therefore, on top of AmbientTalk's passive exception handling system, an active exception handling system is built that features the following requirements to be labelled an ambient oriented exception handling system:

- By relying on futures, the **when-catch** construct allows to not only postpone the execution of code that needs the return value of an asynchronous message, but also to execute handler code as soon as an exception is propagated by the receiver of the message.
- The **group-resolve** construct allows the result of a block of code containing possibly multiple asynchronous invocations to be regarded as a single future. All concurrently raised exceptions in the block are grouped and made available to the programmer such that he can correct the exceptional situation or propagate a single concerted exception.
- To support collaborative exception handling, AmbientTalk provides the **conversation** construct, which allows each participant to be notified of exceptions experienced by their collaboration partners. By specifying collaborative handlers for a given exception, participants of a conversation are able to collaboratively handle an exception.
- Loosely-coupled exception handling is supported through the **due** construct, which allows associating a timeout value with a block of code (possibly containing multiple asynchronous invocations) and raise a predefined exception if messages sent in that code block cannot be delivered in time.

These constructs provide only the strict minimum functionality required to enable ambient oriented exception handling. For example, the **conversation** construct provides no way to encapsulate the collaboration encoded in its body, so exceptions may enter and exit the conversation while it is executing, possibly causing actors not participating in the conversation to be affected. However, higher level constructs, which address this and other issues, can be created using the conversation model. In the next chapter we present some experiments with such higher level constructs.

Chapter 5

Co-ordinated Atomic Actions

5.1 Motivation

Ambient oriented (or by extension any concurrent and distributed) system development is marred by two conflicting phenomena. First of all, ambient oriented software is inherently complex as it deals with a multitude of concurrent, asynchronously communicating devices. Second, the developed software should be dependable, i.e. it reacts in a consistent and acceptable way to errors and failures. Ambient oriented applications do not assume any infrastructure and control about that infrastructure, therefore errors in ambient oriented programs should be anticipated as much as possible. The most beneficial way of achieving such fault tolerance in ambient oriented software is to use system structuring abstractions which have fault tolerance measures associated with them. To ease the reasoning about such complex systems of application logic intertwined with exception handling code, structuring units need to serve as natural areas of error containment and error recovery by restricting somehow interaction and communication of system components. Atomicity of such units is vital for decreasing system complexity when the system exhibits both normal and exceptional behaviour [**?**].

The Co-ordinated Atomic Action (CAA) concept (also discussed in section **??**) was introduced as a unified general approach for structuring complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed object-oriented system. This paradigm provides a conceptual framework for dealing with co-operative and competitive concurrency and for achieving fault tolerance by extending and integrating two complementary concepts - atomic actions (or conversations) and ACID[1] transactions. CAAs provide a mechanism for the strict enclosure of interaction and recovery activities to

[1]Atomicity, Consistency, Isolation and Durability

prevent erroneous information from spreading throughout the whole system.

5.2 Co-ordinated Atomic Actions in AmbientTalk

In this section, we show how CAAs are implemented and used in AmbientTalk. A first observation, is that in AmbientTalk regarding this implementation only actors can be referenced distributedly. This means that we can safely drop the need for transactional semantics to access shared objects, since AmbientTalk simply does not allow sharing passive objects between different actors (there is no competitive concurrency). A second observation we make, related to the first one, is that at the time of writing this dissertation there is no way to guarantee transactional behaviour for actors in AmbientTalk. The reason is that actors may answer messages while their state has not yet been committed, or even send messages in response to a message sent from within the CAA such that these messages may contain tentative data. To prevent this from happening, we restrict the communication of the participants of a CAA such that they cannot send messages to external actors until the CAA is over[2]. Furthermore, all communication between actors happens by means of asynchronous messages. Enabling atomicity of a CAA simply consists of buffering the messages that cross the CAA boundaries and flush them when the CAA is over. This implies that messages that propagate an exception are also confined to the CAA. There are two cases where this happens. The first case is when an actor propagates an exception back to one of the participants of the CAA because some call prior to the participant entering the CAA failed. In this case, the message containing the exception will just be queued until the CAA is over. The other case is when one of the participants itself raises an exception during the execution of the CAA. The CAA may provide its own handler for the exception or make an attempt to rollback the execution and possibly restart the CAA. Rolling back the execution consists of restoring each participant to the state it was in before the CAA started. If the CAA does not handle the exception, the future returned by the invocation of the CAA will be ruined by the exception (the same happens when the CAA throws a different exception while handling an exception) and the caller will have to deal with it. This is different from the classic atomic action model where a general `FailureException` is thrown when an atomic action cannot finish or succesfully rollback the execution. However, in this case where we do not consider the consistency of external shared objects (by limiting communication), buffering the messages that breach the CAA boundary is enough to provide atomicity, and ruining the future returned by a CAA by any exception does not change this. Using futures, CAAs can easily be nested. The enclosing

[2]Currently, a transaction mechanism for actors is being implemented but this was not yet considered during the writing of this dissertation.

CAA will only finish when the futures of its internally started CAAs have been resolved.

The **CAA** construct which we propose here is strongly inspired by the COALA language [?] and looks as follows:

```
CAA(
 participants({
   actor1 => role1;
   actor2 => role2;
   '...More actor-role mappings... '
 }),
 roles({
   role.role1() :: {
     '...Role behaviour... '
    handler('a role handler');
    handler('another role handler');
     '... '
   };
   role.role2() :: {
     '...Role behaviour... '
   };
   '...More roles... '
 }),
 { 'CAA body'
   '... '
 },
 exceptions({
   CaaException1: '... ';
   CaaException2: '... ';
   '...More CAA exceptions... '
 }),
 resolution(
   concert([ caaException1, caaException2 ],
     ConcertedException,
     catch(ConcertedException, 'handler code')
   ),
   '...More exception resolutions... '
 )
)
```

The **participants** construct passed as first argument, maps each participating actor to a role name. This way, participants can be referred to from within the CAA by their corresponding role name.
The second argument is the **roles** construct. It lists the definitions of the roles. By writing a definition of the form **role.**`roleName()`, a role object is created, whose behaviour is temporarily added to the corresponding participant (as specified in the **participants** construct) for the duration of the CAA. This way, it is

possible to specify additional behaviour for a participant in a CAA, including exception handlers.
The third argument is the block of code that will form the body of the CAA.
The fourth argument is the **exceptions** construct, which expects a block of code where exception objects may be defined. These exceptions will be visible to all participants for the duration of the CAA such that they can specify handlers for them in their role definitions.
Finally, the last argument is an exception resolution mechanism. The mechanism we propose here, expects the programmer to list combinations of (possibly concurrently raised) exceptions with for each combination a concerted exception which will be automatically raised when all the exceptions in the combination are encountered during the execution of the CAA. Optionally, a (global) handler can be specified for a concerted exception such that it may be handled by the CAA. If this is not the case, the concerted exception is propagated to each participant, which should have their own role-specific handlers to collaboratively handle the exception.

In the remainder of this section we show how co-ordinated atomic actions can be implemented in AmbientTalk by using the ambient oriented exception handling constructs discussed in section **??**. For the complete source code we refer to appendix **??**.

5.2.1 Starting the conversation

The CAA implementation obviously relies on the **conversation** construct to allow collaborative exception handling. Recall from section **??** that each actor willing to participate in a conversation must implement a **startConversation** method. To produce a language variant which allow actors to participate in a CAA, the original **actor** primitive is shadowed as follows:

```
actor(obj): super().actor(extend(obj, {
  startConversation(conv) :: {
    fut: conv#getRoleBehaviour(thisActor());
    when(fut, becomes(participantsAndBehaviour) -> {
      mixin(this(), participantsAndBehaviour[2]);
      participants := participantsAndBehaviour[1];
      "roleReady"
    })
  }
}));
```

In this case, this method is used to do the necessary initialisation before starting the CAA. First of all, the participant must be extended with the additional behaviour defined in its role. Secondly, each participant must know the other participants. By sending the `getRoleBehaviour` message to the CAA actor (the conversation), the participant receives a table that contains all the participants of the CAA and the role behaviour. Using the `mixin` primitive, the role behaviour is ”mixed in” the current actor behaviour. Each role contains a `participants` instance variable which denotes the CAA participants. This variable is assigned here with the list of participants received from the CAA. As soon as all the initialisation work is done, the participant signals to the CAA actor that it is ready by resolving the future of the `startConversation` message with the string `"roleReady"`.

5.2.2 Mapping participants to roles

The CAA contains a map `participant2RoleMap` that maps each participant to a role name and a role object containing additional behaviour for the participant during the CAA. The **participants** construct is defined as follows:

```
participants( bindings( participant=>role() ) ) :: {
 bindings( participant2RoleMap.put(
   participant, [' RECORD '
     ' ROLE NAME : ' role[1,3,1],
     ' BEHAVIOUR : ' void
   ]) )
};
```

This code excerpt makes extensive use of an AmbientTalk feature called functional parameters. A full discussion of this feature can be found in appendix **??**. For each participant-role binding, a new entry is put in the `participant2RoleMap` whose key is the participant, and whose value is a record containing the name of the role and the behaviour of the role (which will be later filled in with a role object), which is initially an empty vector. The **participants** construct consists of two layers. It takes a code block (denoted by the functional parameter `bindings`) as argument wherein each participant is mapped to a role by the => functional parameter, which in its turn takes a participant actor as argument (left hand operator) and a second functional parameter containing the role reference (right hand operator). The actual name of the reference is extracted from the role reference entity using AmbientTalk's table based reflective interface (see appendix **??** for the reification of language entities).

5.2.3 Defining roles

The **roles** construct makes sure that all role behaviours and their associated handlers are filled in the `participant2RoleMap`. The construct looks as follows:

```
roles( roledefinitions( role(method) ) ) :: {
  '...default role behaviour...'
}
```

The `roledefinitions` functional parameter denotes a closure in whose environment roles may be defined (see appendix **??** for a thorough explanation). Each definition of the form **role**.`method()::{...}` will result in a role object which contains both the behaviour specified in the definition of the role and the default role behaviour prescribed in `role`'s environment. This behaviour is filled in the `participant2RoleMap` by calling the following auxiliary function defined in the local scope of the construct:

```
fillBehaviours(name, behaviour) :: {
  theRoles : participant2RoleMap.getValuesVector();
  theRoles.iterate(lambda(record) -> {
    if(record[1] ~ name,
      if(record[2] ~ void,
        record[2] := behaviour,
        error("Role Behaviour has multiple definitions")))
  })
};
```

Each time a role is created (using **role**.`roleName()::{...}`), this function is called. The correct entry is looked up in the map using the name of the role and when found, its behaviour is filled in. A role behaviour may only be defined once.

Given these two functions, `roledefinitions` is applied on the following behaviour:

```
roledefinitions( {
  ' method is supposed to evaluate to an object '
  roleBehaviour : method();
  roleBehaviour := extend(roleBehaviour, {
    oldDct  : void;
    oldOutbox : void;
    oldInbox : void;

    participants : void;
    blockedMessages : vector.new();
    blockedMailboxes : vector.new();

    roleHandlers : vector.new();

    getCAA() : participants.get(participants.size());

    save() :: {
      oldDct := copy(undoMixin());
      oldOutbox := copy(outbox);
      oldInbox := copy(inbox)
    };

    restore() :: {
      become(oldDct);
      outbox := oldOutbox;
      inbox := oldInbox
    };

    add(mbx, msg) :: {
      if ((participants.contains(msgSource(msg)) &
          participants.contains(msgTarget(msg))), {
        super().add(mbx, msg)
      }, {
        blockedMessages.add(msg);
        blockedMailboxes.add(mbx)
      })
    };

    handler(h): roleHandlers.add(h);

    CAAStarted(caaFuture) :: {
      whenArgs: append([ caaFuture,
        lambda(result) -> void
      ], roleHandlers.asTable());
      fut: when@whenArgs;
      when(fut, becomes(result) -> {
        'Release blocked messages'
        for(i: 1, i<blockedMessages.size(), i:=i+1, {
          super().add(blockedMailboxes.get(i),
                      blockedMessages.get(i))
        });
        undoMixin(this())
      })
    }
  });

  fillBehaviours(method[1,1], rolebehaviour);

  ' return a mechanism to make a new instance '
  roleBehaviour.new()
} )
```

First of all, the role behaviour is created. This happens by first evaluating the `method` argument which denotes the definition of the role object. The resulting object is extended with a number of definitions required for a participant to participate in the CAA.

One requirement for participating in a CAA, is the ability to rollback the state of the participant, as mentioned in section **??**. Rolling back the state of a participant consists of restoring the behaviour of the participant actor to the state it was in prior to joining the CAA and restoring its `inbox` and `outbox`, since it may have scheduled some messages for execution or sending during the CAA. To provide this functionality, two methods are available. The `save` method will save the current behaviour of the participant in `oldDct` and save the current `outbox` and `inbox` in `oldOutbox` and `oldInbox` respectively. The state of the participant can then be rolled back by invoking the `restore` method, which will restore the behaviour saved in `oldDct` and restore the `outbox` and the `inbox` of the participant saved in `oldOutbox` and `oldInbox` respectively.

Messages that cross the CAA boundaries should be buffered until the CAA is over, i.e. these messages should be prevented from being added to the mailbox they normally would be added to. A way of doing this, is by shadowing the `add` primitive function that adds messages to a mailbox, as shown in the code excerpt above. In the role behaviour, we keep a vector `participants` of all the participants including the CAA actor itself to be able to tell the difference between messages that have to be buffered or messages that should be added immediately. Section **??** shows how the `participants` variable is initialised during the initialisation of the CAA. Furthermore, we shadow the original `add` function such that messages are first checked to be internal to the CAA (by checking if source and target are participants of the CAA). If this is the case, the message may safely be added. If this is not the case, the message is buffered such that it can be added to the correct mailbox when the CAA is over.

Inside the role code, the **handler** construct can be called, which is actually just a function that will add a handler to the list of role-specific handlers. All the handlers in this list are registered with the CAA future to allow the participant to handle the corresponding exception (possibly collaboratively with the other participants) when it is propagated by the CAA.

When all participants have been initialised with their corresponding roles (see section **??**) and the CAA is started, each participant needs to be notified, such that it can register its exception handlers on the future returned by the execution of the CAA. For this reason, each role contains a `CAAStarted` method that will do just that. When the CAA is over and all eventually raised exceptions are handled, all the buffered messages are released (i.e. the effects of the atomic action are made visible to the outside world) and the extra role behaviour is removed from the participant actor using the `undoMixin` primitive.

Finally, when the role behaviour is extended with the definitions discussed above, the corresponding table entry in the participant-role mapping is filled in with the resulting role behaviour.

5.2.4 Defining exceptions

CAA exceptions are defined using the **exceptions** construct, which is defined as follows:

```
exceptions(defs()) :: {
  theRoles : participant2RoleMap.getValuesVector();
  theRoles.iterate(lambda(record) -> {
    record[2] := extend(record[2], defs())
  });
  defs()
};
```

All CAA exceptions should be known to each participant such that the participants are able to specify handlers for these exceptions. We chose to extend the role behaviour of each participant with the exception definitions. **exceptions** takes a block of code which will be delayed and wrapped into a closure, since the corresponding parameter `defs()` is a functional one (see appendix **??** for more information regarding functional parameters). This closure is executed when extending the behaviour of each role. Notice that any definition in the block of code passed to **exceptions** will be available to each participant, not only exceptions. However, this is no way to introduce shared objects since these definitions are replicated when sent to each participant (as shown in section **??**). To preserve autonomy of each participant, no behaviour is ever shared.

5.2.5 Defining exception resolutions

Exception resolutions are defined using the **resolution** construct, which is defined as follows:

```
resolution@args :: args;
```

This does nothing more than take an arbitrary number of arguments and return them in a table.
To define a single resolution the **concert** construct is used, which is defined as follows:

```
concert@args :: {
  if (size(args) < 2, {
    error("Concert expects at least two arguments")
  },
  args)
};
```

concert just returns its arguments in a table. When less than two arguments are given, an error is returned since at least a table of exceptions raised in the CAA and the concerted exception should be given. Optionally, a handler can be specified for the concerted exception such that it is immediately handled by the CAA after raising it. In the next section we show how the resulting table from the **exceptions** construct is processed to concert (possibly concurrent) exceptions in the CAA and possibly handle them.

5.2.6 Executing the CAA

As mentioned in section **??**, the **CAA** construct takes five arguments:

```
CAA(participant2Roles,
    roles,
    code(),
    exceptions,
    exceptionRules) :: {
  'CAA implementation'
}
```

In this section we will discuss the implementation of the CAA conversation actor and how the CAA is actually executed.
First of all, the participating actors need to be extracted from the participant-role mapping such that they can be passed to the conversation. This happens as follows:

```
participants: vector.new();
for(i: 1, i<=size(roles), i:=i+1, {
  participants.add(participant2RoleMap.keys().get(i))
});
```

Now we can pass the `participants` variable containing the participating actors to the **conversation** construct as shown below:

```
conv: conversation(participants, groupMixin(futuresMixin({
  participants.add(thisActor());

  '...Rest of the conversation behaviour...'
}
```

Notice that in the **conversation** construct, the conversation actor is added to the participants. This is necessary because the conversation actor (representing the CAA) needs to be able to communicate during the execution of the CAA with the participants playing a role in the CAA. If not added to the `participants` list, the CAA actor itself will be excluded from the interaction by the roles (see section **??**).

To support rollbacks, the conversation has the following method:

```
rollback() :: {
  outbox.clear();
  inbox.clear();
  participants.iterate(lambda(p) -> {
    p#restore()
  })
};
```

All messages scheduled for sending or executing in the CAA actor are cancelled and each participant is sent the `restore` message, which will restore its state to a previously established checkpoint (as discussed in section **??**). This method can be called from within the CAA body or handler or a rollback can be requested by a participant (for example when it catches an exception) by sending the `rollback` message to the CAA.

The CAA contains two pieces of information that a participant needs for participating in the CAA, namely it needs to know the rest of the participants and should receive its role behaviour. As already mentioned in section **??**, participants query the CAA for this information using the `getRoleBehaviour` message, which corresponds to the following CAA method:

```
getRoleBehaviour(participant) :: {
  [ participants, participant2RoleMap.get(participant)[2] ]
};
```

As shown in the code above, the correct role behaviour is looked up in the participant-role map using the participant as key (entry 2 in the table denotes the role object, as explained in **??**) and is returned along with the list of participants in a table.

As already mentioned in section **??**, when each participant is initialised (i.e. it has received the list of participants and its role behaviour from the CAA), the CAA should execute its body:

```
start();
for (i: 1, i<participants.size(), i:=i+1, {
  part: participants.get(i);
  part#CAAStarted(returnedFuture, getHandlers(part))
});
```

As soon as all participants have notified the CAA that they are ready, the CAA is started by calling the function `start` (which is explained below). After that, the `CAAStarted` message is sent to each participant along with the future of the CAA execution (saved in the `returnedFuture` variable). As mentioned in section **??**, this message will cause the participant to register a **when-catch** on the future of the CAA execution with the handlers specified in its role.

The `start` method mentioned above will start the actual execution of the body of the CAA. The method is defined as follows:

```
start() :: {
  for(i:1, i<=size(exceptionRules), i:=i+1, {
    rule: vector.newWithTable(exceptionRules[i][1]);
    resolveRules.add(rule)
  });
  when(,
    group({
      for(i: 1, i<participants.size(), i:=i+1, {
        participants.get(i)#save()
      })
    }, resolve(exc) -> void
  ),
  becomes(saveOk) -> {
    returnedFuture := group(code(), resolve(exceptions) -> {
      'concert/catch/propagate according to CAA exceptionRules'
      for(i:1, i<=resolveRules.size(), i:=i+1, {
        combination: resolveRule.get(i);
        rule: exceptionRules[i];
        if(matches(exceptions, combination),
          if(size(rule) < 3,
            rule[2].throw(),
            try(rule[2].throw(), rule[3])))
      })
    })},
  due(TimeOut, {
    CAAFailedException.new(
          "Participant state save failed").throw())
  }))
};
```

First of all, the exception resolutions are processed. Secondly, a rollback checkpoint is established by sending the `save` message to all participants. When the participants cannot save their state in the prescribed time, the CAA cannot continue and a `CAAFailedException` is thrown. This is encoded by means a **group** which returns a future (representing the combined future of the message sends in its body) that has to be resolved in the time prescribed by the **due** clause of the surrounding **when** construct. As soon as all participants saved their state, the body (wrapped in the `code` closure, see appendix **??**) of the CAA is executed in the **group** clause of a **group-resolve** and all raised exceptions are resolved according to the specified exception resolutions in the **resolve** clause. The resulting future of the **group-resolve** is stored in `returnedFuture`, which will be the return value of the invocation of the **CAA** construct.

5.3 Case Study

In this section we evaluate the usefulness of the **CAA** construct by using it in an implementation of the Ambient Intelligent travel agency introduced in the scenario in section **??**.
As described in the scenario, travellers make a daily selection of events they want to participate in. To this end, up to date event lists are sent out by the travel agency at regular time intervals. Travellers make their choice from that list and both the event provider and the travel agency are notified of the traveller booking an event. When the traveller is successfully enrolled in the event, the entry fee of the event is subtracted from the traveller's bank account. The aforementioned steps need to be structured in an atomic interaction since we do not want for example the traveller to pay a fee if the booking of the event failed, nor we want the traveller to interact with other services as his bank account is currently being accessed. The same holds for the event provider, as long as the process of enrolling a client in the event it provides has not finished (or failed), it remains unclear if another client will be able to join the event because it may be full. Therefore we implement the interaction between the travel agency, the traveller and the event provider as a co-ordinated atomic action. Possible failures include long-lasting disconnections between the three parties and the event that is already at its maximal capacity such that the traveller is not allowed to book that event. Furthermore, a traveller may attempt to book an event that is not available any more or an event which is too expensive.

We will show how the enrolment of a traveller in an event happens using a CAA. The traveller actor is defined as follows:

```
travellerActor(id, agency, initCredits) :: actor(object({
  id_: id;
  agency_: agency;
  eventProviders_: vector.new();
  bookedEventProviders_: vector.new();
  credits_: initCredits;

  updateEventProviders(ep) :: eventProviders_ := ep;

  bookEvent(eventProvider) :: {
    '...CAA...'
  }
}))
```

A traveller actor books an event by calling the `bookEvent` method with the corresponding `eventProvider`, which he finds in his event list. `bookEvent` first defines the following exception and handler to easily reuse them in the CAA specification:

```
BookingFailedException: extend(Exception, {
 type: "BookingFailedException"
});

BookingFailedHandler: catch(BookingFailedException, {
 rollback();
 currentException.throw()
});
```

The `BookingFailedException` will be used to denote a fault where the CAA cannot recover from, in this case it is used to denote that one or more parties have become unreachable. The corresponding handler will just rollback the CAA and rethrow the exception such that the parties still connected may undertake the appropriate measures.

Now we will discuss one-by-one the different components of the CAA. For the complete source code of this toy example (including the definitions of the other actors) we refer to appendix **??**.

```
participants({
 agency_ => agency,
 thisActor() => client,
 eventProvider => provider
})
```

The travel agency actor is mapped to the `agency` role, the traveller actor itself to the `client` role and the event provider actor to the `provider` role. These roles are defined as follows:
The agency role:

```
role.agency() :: {
 isEventProvider(ep) :: {
   if(eventProviders_.contains(ep),
     true,
     UnknownEventException.new("Event not available").throw()
   )
 };

 handler(catch(UnknownEventException, {
   client#updateEventProviders(eventProviders_)
 }))
};
```

The agency role provides the `isEventProvider` to check if an event is valid or available. If not, an `UnknownEventException` is thrown.
The raising of an `UnknownEventException` implies that the traveller had an outdated event list and attempted to book an event that is not available any more. The travel agency handles this exception by explicitly sending a message to the client to update its event list.

The client role:

```
role.client() :: {
  withdraw(amount) :: {
    if (amount > credits_, {
      NotEnoughFundsException.new("Not enough funds").throw()
    }, {
      credits_ := credits_ - amount
    })
  };

  handler(catch(EventFullException, {
    display("The event "+currentException.boundTo().getId+
      " is full. Please pick another event.")
  }));

  handler(catch(BookingFailedException, {
    display("Booking event failed. Please try again later.")
  }));

  handler(catch(UnknownEventException, {
    display("Event is not available. Please pick another event.")
  }));

  handler(catch(NotEnoughFundsException, {
    display("You do not have enough funds
      to participate in this event.")
  }))
};
```

The client role provides a method to withdraw the entry fee for the event from the traveller's bank account. If the traveller does not have the necessary funds, a `NotEnoughFundsException` is thrown.
The rest of the client role behaviour specifies handlers for the different exceptions which just notify the user that the booking failed with the reason why.

The provider role:

```
role.provider() :: {
 addParticipant() :: {
  try({
   when(client#withdraw(fee_), becomes(ok) -> {
    deposit(fee_);
    event_.addParticipant(client)
   }, due(timeOut, {
    BookingFailedException.new("Client not responding").throw()
   }))
  }, catch(EventFullException, {
   getCAA()#rollback();
   EventFullException.bindTo(event_).throw()
  }))
 };

 deposit(amount) :: credits_ := credits_ + amount
}
```

The provider role provides a method to add a participant to the event it provides. First it has to withdraw the entry fee from the client's account. If the client does not respond in the prescribed time, a `BookingFailedException` is thrown because the interaction cannot continue with a disconnected client. If the withdrawal is successful, the same amount is deposited on the provider's account and an attempt is made to enroll the participant in the event. If this fails because the event is already at its maximal capacity, the CAA is rolled back (to undo the money transfer) and an `EventFullException` is thrown.

The body of the CAA is defined as follows:

```
{
 when(agency#isEventProvider(eventProvider), becomes(ok) -> {
  when(provider#addParticipant(client), becomes(val) -> void,
  due(timeOut, {
   BookingFailedException.new(
     "Event provider not responding.").throw()
  }))
 }, due (timeOut, {
  BookingFailedException.new("Agency not responding").throw()
 }))
}
```

First of all, the agency is sent a message to make sure the event is available (using the `isEventProvider` message discussed above). If this is the case, the provider is sent the `addParticipant` message discussed above to enrol the client in the event. When one of the two parties receiving requests in this code does not respond in the prescribed time, a `BookingFailedException` is raised to signal that the CAA failed.

Here we list the exceptions that should be known to all the participants of the CAA:

```
exceptions({
 EventFullException: extend(Exception, {
   type: "EventFullException"
 });
 UnknownEventException: extend(Exception, {
   type: "UnknownEventException"
 });
 NotEnoughFundsException: extend(Exception, {
   type: "NotEnoughFundsException"
 });
 BookingFailedException: BookingFailedException
})
```

Finally, we provide the exception resolution rules:

```
resolution(
 concert([ BookingFailedException ],
   BookingFailedException,
   BookingFailedHandler)
)
```

This single rule will just make sure that if any participant of the interaction is unreachable, the CAA is rolled back and the exception is rethrown (we defined the exception and handler used here beforehand such that they can easily be reused here).

5.3.1 Evaluation

Using the **CAA** construct yields a number of advantages. First of all, by executing atomically, it provides a way of restricting interaction between application components such that tentative data is not spread throughout the whole system, leading a whole constellation of components to an exceptional state. This eases the reasoning about such complex concurrent systems in the face of exceptions and faults. Furthermore, because of the way exception handling is integrated with the construct, exception handling code is separated from regular application logic code. The exception definitions provided in the **exceptions** construct are a good example of this. Without such support, each participant needs to have his own set of exception definitions (`EventFullException`, `UnknownEventException`, `NotEnoughFundsException` and `BookingFailedException` in this case) that must correspond to the exceptions thrown by the other participants or the CAA to be able to handle them, leading to unnecessary code duplication. Furthermore, all behaviour defined in the role objects is specific to the interaction encoded by the CAA (`withdraw` for the traveller, `addParticipant` and `deposit` for the event provider), including all the role-specific handlers to collaboratively handle exceptions. In short, CAAs provide encapsulation of interactions such that the reasoning about their effects is eased and provide a higher degree of separation of concerns with respect to exception handling.

5.4 Conclusion

One of the problems that have to be dealt with when developing ambient oriented applications is mastering the sheer complexity of encoding multiple spontaneous and concurrent interactions between autonomous and distributed application components while remaining fault tolerant. One way of reducing this complexity is by structuring collaborations as co-ordinated atomic actions. In this chapter we have shown by means of a case study that co-ordinated atomic actions can be useful in ambient oriented applications. In fact, CAAs provide encapsulation of interactions such that the reasoning about their effects is eased and provide a higher degree of separation of concerns with respect to exception handling.

The implementation of co-ordinated atomic actions presented in this chapter relies on lower level exception handling constructs. In fact, the **CAA** construct blurs the distinction between a language feature, a reusable high-level structuring mechanism and a design pattern. It solves the common problem of providing atomicity integrated with exception handling in ambient oriented applications, but the way the solution is implemented and used depends on the level of abstraction. We opted for a high level approach to show that using AmbientTalk's minimal ambient oriented exception handling constructs, higher level mechanisms solving different problems (in this case atomicity) can be implemented. Different applications may require more specific mechanisms, but given an ambient oriented exception handling mechanism in combination with a language supporting the ambient oriented programming paradigm, it should be feasible to implement any exception handling behaviour required.

Chapter 6

Conclusion

In the past decade, mobile technology has become increasingly accessible and ever more potent. The mobile devices carried around today include sheer inlimited storage devices (e.g. iPod media players), cell-phones allowing worldwide access to the internet, PDAs with ever increasing processing power, etc. Some of these devices can already communicate over short-ranged wireless protocols such as bluetooth. Ambient Intelligent applications, which consist of such spontaneously interacting autonomous devices in mobile networks, have to deal with the hardware characteristics of these mobile networks, including connection volatility. These hardware characteristics have significant repercussions on the development of Ambient Intelligent applications. Given the added complexity and failure rate of concurrent and distributed applications, exception handling is one of the fields of Ambient Intelligent software development that requires careful investigation.

In Chapter **??** we have described existing exception handling mechanisms and illustrated how these fall short when applied in an ambient oriented context. They either lack support for allowing exceptions to be propagated between different asynchronously executing processes which must be able to handle them in the right context, or for aggregating concurrently raised exceptions such that they can be collectively handled or concerted to a single exception. Furthermore, when different processes on different devices collaborate, they should be informed when one of their collaboration partners experiences an exception, such that the exception can be collaboratively handled by all participants of the collaboration. Additionally, ambient oriented exception handling should be loosely-coupled such that the autonomy of the devices is preserved (i.e. it should not rely on a centralised node and should be able to discriminate long-lasting from temporary disconnections).

Subsequently we have introduced AmbientTalk, an existing ambient oriented

language with limited primitives for supporting ambient oriented exception handling. Whereas the four primitives offered in the language allow dealing with exceptions in ambient oriented software, they offer only low-level support and how to use them to provide high-level exception handling is unclear. Furthermore, because of the ambient oriented exception handling requirements mentioned above, exceptions may be propagated from one device to another, including to devices that were not *directly* involved in the collaboration where the exception originated. This makes it hard to reason about such systems since a whole constellation of devices can be put in an exceptional state by a single failure. This calls for higher level exception handling constructs that allow structuring and encapsulating the collaboration of a number of devices such that the effects of the collaboration are not witnessed by devices outside of the collaboration until the collaboration is over (or signalled failure).
The construct introduced in this dissertation is a Co-ordinated Atomic Action (CAA). CAAs are a mechanism to recursively structure collaborations among concurrent processes such that the interaction between these processes exhibits transaction-like semantics. This is achieved by making sure that the effects of the interaction are only made visible to the outside world once the interaction is over. Furthermore, exception handling is integrated in the CAA model: CAAs may provide their own exception resolution graphs and exception handlers handlers for the interaction they represent and make sure that exceptions raised by one participant of the interaction are raised in the other participants too to enable collaborative exception handling.

6.1 Contributions

The main technical contribution of this thesis is the design and implementation of the Co-ordinated Atomic Action model based on AmbientTalk's ambient oriented exception handling constructs. The introduced **CAA** construct has the following properties:

- Process interactions are encapsulated such that the effects of the interaction encoded by the CAA are not visible to the outside world until the interaction is over. If the CAA cannot finish, all effects are made undone.
- CAAs provide exception resolution graphs which map a set of exceptions raised (possibly concurrently) during the interaction to a single concerted exception.
- CAAs allow additional behaviour required for the interaction to be defined where the interaction is used, namely in the roles associated with each participant of the interaction. This leads to more extensible programs and less

code duplication and offers a concise way of defining collaborative exception handling behaviour by specifying additional exceptions and handlers for the participants in their roles.

- By integrating the **CAA** construct with the future resolution mechanism of AmbientTalk, applications consisting of complex interactions among concurrent processes can be conceived as nested CAAs. This allows thinking about such complex applications as step-by-step executions of the CAAs of the same nesting level and offers controlled propagation of exceptions from CAAs denoting exception contexts to their enclosing CAA, denoting the enclosing context.

We have evaluated these properties by using the **CAA** construct in a case study which showed that co-ordinated atomic actions provide a concise way of structuring concurrent activities where fault tolerance is crucial by encapsulating the collaboration and integrating it with the handling of exceptions.

6.2 Limitations and Future Work

The implementation of co-ordinated atomic actions presented in this dissertation is more restrictive than the original model in the sense that it allows no communication at all between participants of the CAA and external objects. This is in contrast with the original model where participants are allowed to send requests to external objects. The reason why we impose this restriction is that in the implementation of AmbientTalk used at the time of conducting these experiments, there is no way to undo the effects of cascading message sends which may pass tentative data to external actors, since AmbientTalk actors do not guarantee transactional behaviour and have no commit protocol. However, techniques to solve this do exist. One such technique is the TimeWarp algorithm.

6.2.1 TimeWarp

TimeWarp [**?**] is an algorithm for distributing and parallelising discrete event simulations. Parallelising discrete event simulations is a difficult problem because of the need to synchronise the simulation times of different simulation objects without violating the principle of causality. The TimeWarp algorithm solves this by executing events optimistically and later, if necessary, rolling back events that should not in fact have been executed. A simulation executed using TimeWarp is usually decomposed as a series of objects which communicate by passing messages. In simulation terms, these messages can be thought of as scheduling events which are specified to occur at a particular receiving object at a particular time [?].

To rollback the execution of an event two things must be accomplished: restore the state of the system; and undo the effect of any messages sent by the event. This latter is accomplished by sending anti-messages following the earlier messages. When an anti-message is received, it can cause two effects. If the original message had not as yet been executed (it is queued) then it is annihilated and no further action is taken. If the original message had been executed then it will be rolled back (with possible transmission of further anti-messages), the anti-message and message will ”annihilate” and execution will resume as if the original message (and anti-message) had never been received.

A common technique for restoring the state of objects on a rollback is to take a copy of the object’s state each time it receives a message. Then when a rollback occurs, the appropriate previous state can be copied back into the object. One side effect of this is that each object builds op a queue of old state copies. It is hence necessary to eventually garbage collect these old state copies (this is referred to as ”fossil” collection). This can only be done when it is known that a state copy will never be required for a rollback. This is done by periodically computing a value called the GVT (Global Virtual Time) which is equal to the minimum time of any currently active objects or messages between objects. It is guaranteed by the TimeWarp mechanism that no object will rollback to a virtual time prior to GVT, and hence it is possible to fossil collect all state copies with time stamps less than GVT. GVT is also used to ”commit” other interactions with the outside world.

Good performance by TimeWarp relies on a number of factors. First, the cascades of anti-messages induced by rollbacks must damp out quickly and not continue building. The overheads required to restore state on a rollback also cannot be too expensive. In practice, this means that the cost of recording state changes during forward execution must be low. Given all of this, TimeWarp is capable of parallelising and speeding up problems that are otherwise intractable. It has been recognised ever since the algorithm was originally proposed that TimeWarp is not just an algorithm for parallelising discrete event simulations but is also a general purpose technique for synchronising parallel computation. In this context it can be seen as a generalisation of optimistic commit protocols.

Using TimeWarp, it is possible to enable transactional behaviour for objects. In our case, actors that received a faulty message can restore their state if the effects of the message have to be made undone and can trigger the same effect in all the other actors that were affected by the execution of that faulty message

using anti-messages. This way, actors outside the CAA may receive messages from within the CAA, since the effects of these messages can be made undone should the CAA not be able to commit.
There are two ways to integrate the TimeWarp technique in our CAA mechanism. One way is to install TimeWarp in each participant by defining extra behaviour or overriding existing behaviour. The other way is by introducing a special kind of actors that already support the TimeWarp technique. These actors are allowed to participate in a CAA.

6.2.2 Applications

The CAA model already has shown its merit in classic concurrent and distributed systems. To evaluate it in an ambient oriented context, we used the CAA mechanism in a brief case study. However, it seems natural that the full potential and possible problems requiring adaptation of the mechanism will only be uncovered when it is used in a larger scale setting, since one of the primary goals of the CAA model is to reduce the complexity of interacting concurrent processes in such large scale applications. Therefore, future work remains the development of larger ambient oriented applications using the CAA model to fully understand its applicability in this context.

Appendix A

Exceptions in AmbientTalk: explicit parent and prototype pointers

The highest object in the AmbientTalk exception hierarchy is the `throwable` object. This object always responds with `false` to `canBeHandled`. To ex-

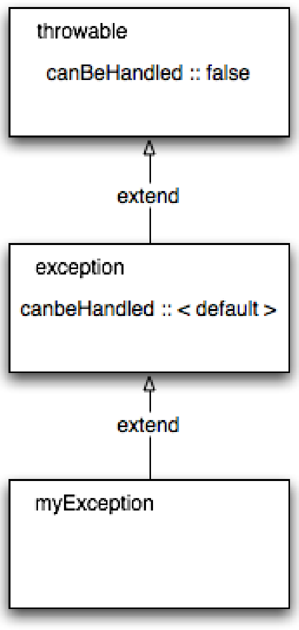

Figure A.1: Exception hierarchy with explicit parent and prototype pointers

plain the exact reason why each exception object needs to be instrumented with a prototype and a parent pointer, consider the following simple hierarchy of exception objects depicted in figure **??**. `myException` extends the standard exception object, which in its turn extends `throwable`. We illustrate two different scenarios:

- A handler that catches `myException` is encountered on the stack after throwing an exception `myException.new()`. To determine whether this handler can be invoked, the `canBeHandled` mechanism needs some way to recognise the relation between both objects. It is not sufficient to compare the parents of both objects, because then this exception may be caught by any handler that catches direct extensions of the `exception` object. Therefore, every exception object has a link to the first prototype (i.e. an extension of parent exception object) from which it was cloned.
- A more general handler catching `exception` is encountered on the stack after throwing an exception `myException.new()`. The `canBeHandled` function first compares the prototypes of the handler's exception and the thrown exception. Since they are not equal, the implementation needs to check whether the handler may catch a supertype of the thrown exception. This cannot be achieved using a `super`-call (since `canBeHandled` is implemented in `exception`, `super` denotes `throwable`). Rather the `canBeHandled` method must be invoked on the parent of the receiving `myException` clone. Therefore this parent must be made accessible in the object.

However, client code that seeks to define new exception types, should not be aware of the pointer plumbing that happens behind the scenes, which is why the `extend` method will take its argument (a code block wrapped in a closure) and use meta-programming operations (table access and constructor functions) to perform these operations entirely behind the scenes. The default `extend` method below shows how two instructions (the definition of the parent of the child exception as the prototype of the current exception and the prototype of the child exception as the environment to be "captured" using `capture`) are added to the body of the closure (accessed by the tabulation `extension[1,3]`) and how this parsetree is used to create a new closure. Also notice that the scope of the closure is set to `this`, which ensures that the closure will be applied in (and thus extends) the receiver object, rather than the call-site.

```
extend(extension())::{
 parsetree: application(reference("begin"), [ extension[1,3],
   application(reference("def"), [ "parentType", "prototype" ]),
   application(reference("def"), [ "prototype", "capture()" ])
 ]);

 'Fill in the real code'
 extension := closure(function("extension" , [], parsetree),
                    this,
                    this);
 extension()
};
```

Given these abstractions (combined with suitable initial values for the `exception` object) the default behaviour of `canBeHandled` can be described as follows :

```
canBeHandled(handlerException)::{
 if((prototype = handlerException.getPrototype()) &
    (boundTo() = handlerException.boundTo()),
   { true },
   { parentType.canBeHandled(handlerException) })
};
```

Appendix B

Language Support for Syntactic Extensions

In this thesis we have introduced the co-ordinated atomic action as an abstraction construct to interactions between autonomous actors. The realisation of this abstraction relies on the introduction of additional syntax which heavily relies on AmbientTalk's language extension support.

B.1 Reification of Language Entities

AmbientTalk language entities are reified in the language as tables containing in the slots the components of the language entity. A function for example, consists of a name, a table of parameters and a body. Consequently, a function is reified in the language as a table containing three slots: the name of the function, its argument table, and its body.

	1	2	3
application	name	arguments	/
assignment	invocation	expression	/
closure	function	dictionary	/
declaration	invocation	expression	/
definition	invocation	expression	/
function	name	parameters	body
message	receiver	invocation	/
reference	name	/	/
tabulation	name	index	/

Table B.1: Language Entities Table Representations

Each of the names in the first column of the table given here represents also a constructor which can be used to create the corresponding language entity. For example, a faculty function can be created as follows:

```
function(fac, [ n ], if(n < 2, 1, (n * fac(n-1))))
```

B.2 Functional Parameters

In AmbientTalk, functions can have two kinds of parameters: normal parameters and functional parameters. The actual kind of a parameter is syntactically visible in the definition of the function. Binding actual arguments to formal parameters corresponds to an implicit definition of the formal parameter to the actual argument. In the case of normal parameters, the argument is evaluated and the associated value is bound to the formal parameter before the body of the function is evaluated. In order to describe the behaviour associated to functional parameters, it is best to look at an example:

```
zero(a, b, f(x), epsilon): {
  c: (a+b)/2;
  if(abs(f(c)) < epsilon,
    c,
    if(f(a)*f(c) < 0,
       zero(a, c, f(x), epsilon),
       zero(c, b, f(x), epsilon)))
}
```

The function `zero` defined above is executed in an environment with four definitions: three variables (`a`, `b`, and `epsilon`) and a function `f` taking a single parameter x. This AmbientTalk function looks for a zero of a function f(x) between a and b given a precision epsilon. The fact that f(x) is a functional argument results in the third expression (upon calling zero) never being evaluated. The third argument is thus interpreted as an expression (depending on x) to be used as the body for a newly defined function called f and one parameter x. Hence, when calling zero(-1, 1, x*2-5, 0.001), the result is that inside zero, four names are accessible: a, b, f and epsilon.

Functional parameters allow us to extend AmbientTalk in itself since they cause their actual arguments to be delayed. The following code excerpt shows how the AmbientTalk while function is programmed in AmbientTalk itself. In the same way, true, false, and, or and if can be implemented as functions that happen to delay some of their arguments:

```
while(cond(),body()): {
  loop(value,pred): pred(loop(body(),cond()),value);
  loop(void,cond())
}
```

The while function assumes the Church encoding of booleans:

```
true(t(), f()): t()
false(t(), f()): f()
```

These examples show how functional parameters (of zero arguments in these cases) allow for automatic thunk creation upon calling a function. This is in contrast to languages like SmallTalk or Self where one has to manually delay arguments by wrapping them in a function.

B.3 Method Attributes

Another mechanism to change the behaviour of a method is to annotate it with a special method attribute. These attributes, which can be defined in the language itself allow modifying the behaviour of a method in a reusable manner. The evaluation semantics of a method attribute is as follows: when evaluating a definition of the form exp.m(args): body, the default behaviour for creating a method (namely a function is created with name m, the arguments args and body) is used. However, the created function is not yet bound in the current dictionary. At this point, the expression exp is expected to evaluate to a function *f* which takes a single parameter: the newly created function m. Rather, m will eventually get bound to the result of applying *f* to the unbound function. Applying *f* to the method m can yield a *function transformation* if *f* is a higher order function that returns another function. This way, methods can be ”wrapped” to allow for reusable syntax elements to be introduced in the language.

B.4 Multi-layered Syntax Extensions

There are situations in which one may want to locally introduce syntax extensions. In this section, we show how this can be done using functional parameters and method attributes by explaining the way how roles are defined in the **CAA** construct (see **??**).
Roles are defined by calling the **roles** construct with a block of code that contains role definitions of the form **role**.`name()`**::** { ... }. **role** is an example of a method attribute as explained in the previous section and as explained in section **??** it ensures that the specified role method returns an object that consists of the behaviour defined in the method body plus the default behaviour exhibited by any role. The `method` parameter (see below for the code) thus denotes the method body which will be the behaviour of the role. However, the role behaviour should also contain the necessary behaviour to allow its corresponding participant to participate in the CAA. Therefore, we apply the **role** method attribute which will perform a function transformation (as explained in the previous section) on the method body such that the resulting behaviour from executing `method()` is extended with the necessary default role behaviour (this is further explained below).
The definition of the **roles** construct which provides a localised definition of the role method attribute looks as follows:

```
roles( roledefinitions( role(method) ) ) :: {
  '...default role behaviour...'
}
```

What happens is that the **role** syntax is introduced locally to the code block denoted by `roledefinitions` by making `roledefinitions` a functional parameter that takes in its turn a functional parameter. This functional parameter denotes the method attribute definition **role** that will be applied on an argument `method`, which denotes the actual behaviour of the role. Thus executing `method()` will just return the behaviour defined for the role. However, roles need additional behaviour to allow their corresponding actors to participate in the CAA. Therefore, the role definitions (each one denoted by a `method` parameter) are wrapped using the **role** function as a method attribute, which is bound to a function that returns a role by extending the behaviour returned by the function it wraps (denoted by the `method` argument) with the default role behaviour.
This means concretely that each occurrence of `method` in the body of `roleDefinitions` should evaluate to an object such that it can be extended

by the **role** function resulting from using it as an attribute to the definition of the `method` object. This way, **roles** will transform each definition of the form **role**`.name():: { ... }` in the passed code block to a role object (with all the necessary behaviour) and insert the role name and behaviour in the correct entry of the participant-role mapping.

Appendix C

CAA Source Code

C.1 Starting the conversation

```
actor(obj): super().actor(extend(obj, {
  startConversation(conv) :: {
    fut: conv#getRoleBehaviour(thisActor());
    when(fut, becomes(participantsAndBehaviour) -> {
      mixin(this(), participantsAndBehaviour[2]);
      participants := participantsAndBehaviour[1];
      "roleReady"
    })
  }
}));
```

C.2 Mapping participants to roles

```
participant2RoleMap: smallmap.new();

participants( bindings( participant=>role() ) ) :: {
  bindings( participant2RoleMap.put(
    participant, [' RECORD '
      ' ROLE NAME : ' role[1,3,1],
      ' BEHAVIOUR : ' void
    ]) )
};
```

C.3 Defining roles

```
roles( roledefinitions( role(method) ) ) :: {
  fillBehaviours(name, behaviour) :: {
    theRoles : participant2RoleMap.getValuesVector();
    theRoles.iterate(lambda(record) -> {
      if(record[1] ~ name,
        if(record[2] ~ void,
          record[2] := behaviour,
          error("Role Behaviour has multiple definitions")))
     })
  };

  roledefinitions( {
    ' method is supposed to evaluate to an object '
    roleBehaviour : method();
    roleBehaviour := extend(roleBehaviour, {
      oldDct  : void;
      oldOutbox : void;
      oldInbox : void;

      participants : void;
      blockedMessages : vector.new();
      blockedMailboxes : vector.new();

      roleHandlers : vector.new();

      getCAA() : participants.get(participants.size());

      save() :: {
        oldDct := copy(this());
        oldOutbox := copy(outbox);
        oldInbox := copy(inbox)
      };

      restore() :: {
        become(oldDct);
        outbox := oldOutbox;
        inbox := oldInbox
      };

      setParticipants(parts) :: participants:=parts;

      add(mbx, msg) :: {
        if (((participants.contains(msgSource(msg)) &
              participants.contains(msgTarget(msg))) |
             (!participants.contains(msgSource(msg)) &
              !participants.contains(msgTarget(msg)))), {
          super().add(mbx, msg)
        }, {
```

```
        blockedMessages.add(msg);
        blockedMailboxes.add(mbx) })
    };

    handler(h): roleHandlers.add(h);

    CAAStarted(caaFuture, handlers) :: {
      whenArgs: append([ caaFuture,
        lambda(result) -> void
      ], handlers.asTable());
      fut: when@whenArgs;
      when(fut, becomes(result) -> {
        'Release blocked messages'
        for(i: 1, i<blockedMessages.size(), i:=i+1, {
          super().add(blockedMailboxes.get(i), blockedMessages.get(i))
        });
        undoMixin(this())
      })
    }
  });

  fillBehaviours(method[1,1], rolebehaviour);

  ' return a mechanism to make a new instance '
  roleBehaviour.new()
 })
};
```

C.4 Defining exceptions

```
exceptions(defs()) :: {
  theRoles : participant2RoleMap.getValuesVector();
  theRoles.iterate(lambda(record) -> {
    record[2] := extend(record[2], defs())
  });
  defs()
};
```

C.5 Defining exception resolutions

```
resolution@args :: args;

concert@args :: {
  if (size(args) < 2, {
    error("Concert expects at least two arguments")
  },
  args)
};
```

C.6 Executing the CAA

```
CAA(participant2Roles, roles, code(), exceptions, exceptionRules) :: {
  participants: vector.new();
  for(i: 1, i<=size(roles), i:=i+1, {
    participants.add(participant2RoleMap.keys().get(i))
  });

  conv: conversation(participants, groupMixin(futuresMixin({
    participants.add(thisActor());

    participantsReady: 1;

    CAAFailedException: extend(Exception, {
      type: "CAAFailedException"
    });

    resolveRules: vector.new();

    returnedFuture: void;

    rollback() :: {
      outbox.clear();
      inbox.clear();
      participants.iterate(lambda(p) -> {
          p#restore()
      })
    };

    getRoleBehaviour(participant) :: {
      [ participants, participant2RoleMap.get(participant)[2] ]
    };

    getHandlers(participant) :: {
      participant2RoleMap.get(participant)[3]
    };

    'Checks whether each exception in exc has a supertype
    in comb. If this is the case, exc matches the exception
    rule comb'
    matches(exc, comb): {
      match: true;
      for(i: 1, i<=exc.size(), i:=i+1, {
        ok: comb.detect(lambda(el) -> {
          exc.get(i).canBeHandled(el)
        });
        if(!ok, match := false)
      });
      match
    };

    start()::{
      for(i:1, i<=size(exceptionRules), i:=i+1, {
        rule: vector.newWithTable(exceptionRules[i][1]);
        resolveRules.add(rule)
      });
      when(,
        group({
          for(i: 1, i<participants.size(), i:=i+1, {
            participants.get(i)#save()
          })
        }, resolve(exc) -> void
      ),
      becomes(saveOk) -> {
        returnedFuture := group(code(), resolve(exceptions) -> {
        'concert/catch/propagate according to CAA exceptionRules'
          for(i:1, i<=resolveRules.size(), i:=i+1, {
            combination: resolveRule.get(i);
            rule: exceptionRules[i];
            if(matches(exceptions, combination),
              if(size(rule) < 3,
                rule[2].throw(),
                try(rule[2].throw(), rule[3])))
          })
        })
      },
      due(TimeOut, {
        CAAFailedException.new(
          "Participant state save failed").throw())
      }))
    };

    start();

    capture()
  })));

  returnedFuture
};
```

Appendix D

Case Study Source Code

D.1 Event Object

```
makeEvent(id, capacity, fee) :: object({
  id_: id;
  capacity_: capacity;
  fee_: fee;
  participants_: vector.new();

  EventFullException: extend(Exception, {
    type: "EventFullException"
  });

  getProvider() :: provider_;

  addParticipant(participant) :: {
    if (participants.size() = capacity_, {
      EventFullException.new("Event "+id+" full").throw()
    }), {
      participants_.add(participant)
    }
  }
})
```

D.2 Event Provider Actor

```
eventProviderActor(id, agency, event) :: actor(object({
  id_: id;
  agency_: agency;
  event_: event;
  credits_: 0;
```

```
 getEvent() :: event_;
 getFee() :: fee_
}))
```

D.3 Agency Actor

```
agencyActor(id) :: actor(object({
 id_: id;
 eventProviders_: vector.new();
 travellers_: vector.new();

 addEventProvider(ep) :: eventProviders_.add(ep);
 removeEventProvider(ep) :: eventProviders_.remove(ep)
}))
```

D.4 Traveller Actor

```
travellerActor(id, agency, initCredits) :: actor(object({
 id_: id;
 agency_: agency;
 eventProviders_: vector.new();
 bookedEventProviders_: vector.new();
 credits_: initCredits;

 updateEventProviders(ep) :: eventProviders_ := ep;

 bookEvent(eventProvider) :: {
    BookingFailedException: extend(Exception, {
     type: "BookingFailedException"
    });
    BookingFailedHandler: catch(BookingFailedException, {
     rollback();
     currentException.throw()
    });
    CAA(
     participants({
      agency_ => agency,
      thisActor() => client,
      eventProvider => provider
     }),
     roles({
      role.agency() :: {
       isEventProvider(ep) :: {
```

```
      if(eventProviders_.contains(ep),
        true,
        UnknownEventException.new("Event not available").throw()
      )
    };

    handler(catch(UnknownEventException, {
      client#updateEventProviders(eventProviders_)
    }))
  };

  role.client() :: {
    withdraw(amount) :: {
      if (amount > credits_, {
        NotEnoughFundsException.new(
          "Not enough funds").throw()
      }, {
        credits_ := credits_ - amount
      })
    };

    handler(catch(EventFullException, {
      display("The event "+currentException.boundTo().getId+
        " is full. Please pick another event.")
    }));

    handler(catch(BookingFailedException, {
      display("Booking event failed.
        Please try again later.")
    }));

    handler(catch(UnknownEventException, {
      display("Event is not available.
        Please pick another event.")
    }));

    handler(catch(NotEnoughFundsException, {
      display("You have not enough funds
        to participate in this event.")
    }))
  };

  role.provider() :: {
    addParticipant() :: {
      try({
        when(client#withdraw(fee_),
        becomes(ok) -> {
          deposit(fee_);
          event_.addParticipant(client)
        }, due(timeOut, {
          BookingFailedException.new(
            "Client not responding").throw()
        }))
      }, catch(EventFullException, {
        getCAA()#rollback();
        EventFullException.bindTo(event_).throw()
      }))
    };

    deposit(amount) :: credits_ := credits_ + amount
  }
}),

{
 when(agency#isEventProvider(eventProvider), becomes(ok) -> {
   when(provider#addParticipant(client), becomes(val) -> void,
   due(timeOut, {
     BookingFailedException.new(
       "Event provider not responding.").throw()
   }))
 }, due (timeOut, {
   BookingFailedException.new("Agency not responding").throw()
 }))
},

exceptions({
  EventFullException: extend(Exception, {
    type: "EventFullException"
  });
  UnknownEventException: extend(Exception, {
    type: "UnknownEventException"
  });
  NotEnoughFundsException: extend(Exception, {
    type: "NotEnoughFundsException"
  });
  BookingFailedException: BookingFailedException
}),

resolution(
  concert([ BookingFailedException ],
    BookingFailedException,
    BookingFailedHandler),
  concert([ BookingFailedException,
            BookingFailedException ],
    BookingFailedException,
    BookingFailedHandler),
  concert([ BookingFailedException,
            BookingFailedException,
                BookingFailedException ],
            BookingFailedException,
            BookingFailedHandler)
        )
      )
  }
}))
```

Bibliography

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[Agh90] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[AH86] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. *Massachusetts Institute Of Technology Artificial Intelligence Laboratory*, 1986.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[BK03] Peter A. Buhr and Roy Krischer. Bound exceptions in object programming. *European Conference on Object-Oriented Programming*, 2003.

[BM00] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng.*, 26(9):820–836, 2000.

[Bri88] Jean-Pierre Briot. From objects to actors: Study of a limited symbiosis in smalltalk-80. *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, September 1988.

[BRR$^+$00] D. M. Beder, A. Romanovsky, B. Randell, C. R. Snow, and R. J. Stroud. An application of fault tolerance patterns and coordinated atomic actions to a problem in railway scheduling. *SIGOPS Operating Systems Review*, 34(4):21–31, 2000.

[BRRR01] Delano M. Beder, Brian Randell, Alexander Romanovsky, and Cecilia. M.F. Rubira. On applying coordinated atomic actions and dependable software architectures for developing complex systems. *International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE*, 4, 2001.

[BS95] David E. Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, 1995.

[BSd82] Roy J. Byrd, Stephen E. Smith, and S. Peter deJong. An actor-based programming system. In *Proceedings of the SIGOA conference on Office information systems*, pages 67–78, New York, NY, USA, 1982. ACM Press.

[CC05] Denis Caromel and Guillaume Chazarain. Robust exception handling in an asynchronous environment. In *ECOOP Workshop on Exception Handling in Object-Oriented Systems: Developing Systems that Handle Exceptions*, 2005.

[CPK95] J. G. Cleary, M. Pearson, and H. Kinawi. The architecture of an optimistic cpu: the warpengine. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 163, Washington, DC, USA, 1995. IEEE Computer Society.

[DB04] Jessie Dedecker and Werner Van Belle. Actors for mobile ad-hoc networks. *Lecture Notes in Computer Science*, 3207:482–494, August 2004.

[DCM$^+$05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient oriented programming. *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-Oriented programming systems, languages and applications, ACM Press*, 2005.

[Don01] Christophe Dony. A fully object-oriented exception handling system: rationale and smalltalk implementation. *Advances in exception handling techniques*, pages 18–38, 2001.

[DVM$^+$] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. *European Conference on Object-Oriented Programming*.

[EM97] W. Keith Edwards and Elizabeth D. Mynatt. Timewarp: techniques for autonomous collaboration. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 218–225, New York, NY, USA, 1997. ACM Press.

[Goo75] John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[IR05] Alexei Iliasov and Alexander Romanovsky. Exception handling in coordination-based mobile environments. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, pages 341–350, Washington, DC, USA, 2005. IEEE Computer Society.

[Iss93] Valérie Issarny. An exception handling mechanism for parallel object-oriented programming: towards the design of reusable, and robust distributed software. *Journal of Object-Oriented Programming*, 1993.

[Iss01] Valérie Issarny. Concurrent exception handling. pages 111–127, 2001.

[iTY00] Shin ichi Tazuneki and Takaichi Yoshida. Concurrent exception handling in a distributed object-oriented computing environment. In *ICPADS '00: Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, page 75, Washington, DC, USA, 2000. IEEE Computer Society.

[IY91] Yuuji Ichisugi and Akinori Yonezawa. Exception handling and real time features in an object-oriented concurrent language. In *Proceedings of the UK/Japan workshop on Concurrency : theory, language, and architecture*, pages 92–109, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[KBS$^+$01] K.Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J-C. Burgelman. Istag scenarios for ambient intelligence in 2010. *Information Society Technologies Advisory Group, European Commission*, 2001.

[Kin] Joseph R. Kiniry. Exceptions in java and eiffel: Two extremes in exception design and application.

[KRS01] Jörg Kienzle, Alexander Romanovsky, and Alfred Strohmeier. Open multithreaded transactions: Keeping threads and exceptions under control. In *6th International Workshop on Object-Oriented Real-Time Dependable Systems, Roma, Italy, 8 - 10 January, 2001*, pages 197–205, Los Alamitos, California, USA, 2001. IEEE Computer Society Press.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages* *and applications*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[MDB$^+$] Stijn Mostinckx, Jessie Dedecker, Elisa Gonzalez Boix, Tom Van Cutsem, and Wolfgang De Meuter. Ambient-oriented exception handling.

[Mey88] Bertrand Meyer. Disciplined exceptions. Technical report, Goleta, CA, 1988.

[Mil04] Robert Miller. The guardian model and primitives for exception handling in distributed systems. *IEEE Trans. Softw. Eng.*, 30(12):1008–1022, 2004. Member-Anand Tripathi.

[MPR01] A. Murphy, G. Picco, , and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 524, Washington, DC, USA, 2001. IEEE Computer Society.

[MT02] Robert Miller and Anand Tripathi. The guardian model for exception handling in distributed systems. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 304, Washington, DC, USA, 2002. IEEE Computer Society.

[MTS05] Mark S. Miller, Eric Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in e as plan coordination. In *TGC*, pages 195–229, 2005.

[MZL02] M. Mamei, F. Zambonelli, and L. Leonardi. Programming context-aware pervasive computing applications with tota, 2002.

[MZL03] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 342, Washington, DC, USA, 2003. IEEE Computer Society.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. Lime: Linda meets mobility. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 368–377, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[SDUV] Frederic Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Improving exception handling in multi-agent systems.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242, 1991.

[VG00] Julie Vachon and Nicolas Guelfi. COALA: a design language for reliable distributed system engineering. In *Workshop on Software Engineering and Petri Nets*, pages 135–154. DAIMI, June 2000.

[Wei93] Mark Weiser. Ubiquitous computing. *IEEE Computer Hot Topics*, 1993.

[Wei95] Mark Weiser. The computer for the 21st century. pages 933–940, 1995.

[XRR98] Jie Xu, Alexander B. Romanovsky, and Brian Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *International Conference on Distributed Computing Systems*, pages 12–21, 1998.

[Xu88] A. S. Xu. A FAULT-TOLERANT NETWORK KERNEL FOR LINDA. Technical Report MIT/LCS/TR-424, 1988.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming abcl/1. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268, New York, NY, USA, 1986. ACM Press.

[ZS99] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 435–446, New York, NY, USA, 1999. ACM Press.