Ambient-Oriented Exception Handling (DRAFT)[⋆]

Stijn Mostinckx[⋆⋆], Jessie Dedecker[⋆⋆⋆],
Elisa Gonzalez Boix, Tom Van Cutsem[⋆⋆], and Wolfgang De Meuter
`smostinc | jededeck | egonzale | tvcutsem | wdmeuter@vub.ac.be`

Programming Technology Laboratory
Vrije Universiteit Brussel, Belgium

Abstract. Writing ambient-oriented software for mobile devices connected through wireless network connections provides a new challenge in the field of exception handling. It involves dealing with issues such as asynchronous communication, moving hardware and software, only to name a few. Building on an analysis of the fundamental differences between mobile networks and their stationary counterparts, this paper establishes a set of criteria for an ambient-oriented exception handling mechanism. We subsequently present ambient conversations, a novel distributed exception handling mechanism that adheres to the prescribed criteria, and describe its realisation in the ambient-oriented programming language AmbientTalk.

1 Introduction

Recently, mobile hardware has evolved to become ever cheaper, smaller and yet more powerful. The current crop of mobile devices have wireless network provisions (such as Bluetooth and WiFi) that allow them to collaborate with one another in open, mobile and highly dynamic networks. Such networks of mobile devices are crucial to achieve Weiser's vision of ubiquitous computing [24], where computers are seamlessly integrated in the fabric of everyday life. Similarly, the spontaneous interaction between both mobile and embedded devices plays an important role in the various Ambient Intelligence (AmI) scenarios proposed by the IST Advisory Group of the European Union [7].

At present, little experience exists in developing applications that fully exploit the potential of mobile networks of wirelessly connected devices. A significant obstacle impeding the construction of such applications is the fact that contemporary programming languages lack proper support to deal with the specific

[⋆] This paper is a draft as submitted for review. The final version was published in *Advanced Topics in Exception Handling Techniques*, eds. C. Dony, J. L. Knudsen, A. Romanovsky, A. Tripathi, Springer-Verlag as part of the Lect. Not. Comp. Sci. series, no. 4119 (2006)

[⋆⋆] Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

[⋆⋆⋆] Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

properties that distinguish mobile networks from their stationary counterparts. The most important of these properties are that disconnection is the rule rather than the exception (because the communication range of wireless technology is limited) and that the network is open (because devices can appear and disappear unheraldedly).

The goal of our work is to uncover the necessary language abstractions and implementation techniques that ease the development of applications which are based on spontaneous interaction between different mobile devices. Since little software engineering experience exists in developing such systems, our research is based on a thorough analysis of the hardware phenomena that fundamentally distinguish mobile from stationary networks. In previous work, this analysis led us to propose a number of fundamental programming language characteristics that define a novel ambient-oriented programming (AmOP) paradigm [5]. This paradigm – based on asynchronously communicating actors provided with innate mechanisms for discovering services in the ambient (*i.e.* its immediate environment) as well as reversible computing – was embodied in the AmbientTalk language [4].

This paper focusses on the issue of handling exceptions which occur during such spontaneous interactions of mobile devices. The next section establishes the essential differences between mobile and stationary networks and illustrates how they affect an exemplar collaborative editing application deployed on mobile hardware. Subsequently, we derive a set of criteria for future ambient-oriented exception handling mechanisms and relate them to the example application (section 3). Section 4, provides an overview of existing distributed exception handling techniques and evaluates them with respect to the proposed criteria. Based on this analysis of the (shortcomings of) existing approaches, section 5 proposes ambient conversations as a first tentative ambient-oriented exception handling mechanism.

2 Motivation

As mentioned in the introduction, our goal is to cater for spontaneous interaction between different mobile devices. The hardware properties of these mobile devices engender a number of phenomena that need to be dealt with when writing ambient-oriented software. These phenomena have led us to propose a novel ambient-oriented programming paradigm tailored toward prescribing such spontaneous interactions of mobile devices [5]. This section recapitulates the hardware phenomena that lie at the basis of the ambient-oriented programming paradigm and illustrates the repercussions of the various hardware phenomena on an exemplar collaborative editing application.

2.1 Hardware Phenomena

With the current state of commercial technology, mobile devices are often characterised as having scarcer resources (such as lower CPU speed, less memory and limited battery life) than traditional hardware. However, in the last couple of years, mobile devices and full-fledged computers like laptops are blending more and more. That is why we do not consider such restrictions as fundamental as the following phenomena which are inherent to networks of mobile devices:

Connection Volatility. Collaborating devices cannot assume a stable connection due to the limited communication range of the wireless technology combined with the fact that users can move out of range. Therefore, disconnection of a device should not necessarily be interpreted as a failure. Often, the task is expected to resume when the disconnected party returns within a reasonable amount of time. An alternative might be to continue the interaction with a replacement service. These examples merely serve to illustrate that ambient-oriented software is typically expected to perform its task in the presence of volatile connections.

Ambient Resources. When developing software for mobile devices, it is important to realise that, as the device roams, new resources will become dynamically (un)available in its ambient. It is therefore unrealistic to encode knowledge about the availability of a service explicitly (*e.g.* storing a server address). Instead a mechanism is needed to dynamically manage the set of ambient resources.

Autonomy. Mobile devices should be able to act as autonomous computing devices. First of all, this implies that it should be possible for a device to collaborate with other devices, without requiring to be connected to infrastructure (*e.g.* a server) to either discover other participants or to coordinate the interaction. Similarly, the device must be able to recover when one of its communication partners disconnects, such that the device does not remain blocked until that communication partner returns.

Natural Concurrency. The autonomy phenomenon clearly stimulates different devices to collaborate in an entirely concurrent fashion. Typically these concurrent devices communicate using asynchronous communication since waiting for the result of synchronous invocations undermines the autonomy of the waiting device. When specifying collaborations between different mobile devices, it is necessary to orchestrate the naturally concurrent devices.

In previous work, the implications of these hardware phenomena on the design of programming languages have been thoroughly analysed [5]. To enhance the construction of applications for networks of mobile devices, the *Ambient-Oriented Programming* paradigm was established. Languages adhering to this paradigm have distribution characteristics which are designed with respect to the hardware phenomena summarised above. First of all, ambient-oriented programming languages break with the tradition of structuring programs using classes which are instantiated at run-time into objects. Rather, the these languages are object-based: code and data are packed together directly in an object, rather than storing the code separately in a class. When distributing objects across a network, this has proven to be a more flexible approach and forgoes much of the problems related to class versioning [23]. Secondly, ambient-oriented programming languages must safeguard the autonomy of all concurrent processes. That is to say that no process should be blocked when one of its communication partners goes out of earshot. This property can be achieved when the language's concurrency model relies solely on *non-blocking communication primitives*. Thirdly, ambient-oriented programming languages facilitate collaborating processes to continue working to preserve their autonomy. This may yield inconsistencies between the different processes. To resolve these inconsistencies, processes store an *explicit representation* (i.e. a reification) of their communication details. Such *reified communication state* allows a process to properly recover from an inconsistency by reversing (part of) its computation. Finally, ambient-oriented programming languages need to provide support for distributed naming [6], to dynamically discover ambient resources without knowing their address beforehand.

This paper provides a similar analysis of the implications of the hardware phenomena on the design of an exception handling mechanism. Whereas an ambient-oriented programming language will provide some rudimentary exception handling support by means of its *reified communication traces*, an additional mechanism is clearly needed to be able to handle both functional exceptions (which are raised explicitly) as well as non-functional ones (*e.g.* long-lasting disconnections). To ground the discussion, the remainder of this section describes an illustrative example application which highlights the effects of the hardware phenomena on both the application logic and on how one expects exceptions to be raised, propagated and handled in the system.

2.2 A Sample Application: Collaborative Editing

When developing ambient-oriented applications, the effect of hardware phenomena described above permeates the entire application. This is illustrated in this section using a classic distributed application, namely a collaborative editor. The application consists of participant editors which are distributed over various mobile devices. Each participant has its own copy of a document which is synchronised regularly with the versions of the other editors. To create a collaborative editing session, a single editor may publish a document, and specify which users may join to edit the document. These users will receive an invitation containing a copy of the document. The way the hardware phenomena influence the construction of such an editor is discussed next.

Connection Volatility Editors should not be excluded from the collaboration when they go out of communication range. Despite being disconnected, the editor may continue making local changes. When reconnecting within a reasonable amount of time, the changes made locally are synchronised and the editor is once again working on the same document as the other participants.

Ambient Resources In networks of mobile and frequently disconnecting devices, invitations are preferably issued based on high-level information such as *e.g.* the user's name rather than low-level network addresses. This makes it possible to invite users whose devices are currently not reachable or whose device's address is not known (the address of a mobile device may change frequently). The ambient resource management is responsible for spontaneously discovering devices based on such high-level information. Once such a device is found, it can be provided with an initial version of the document.

Autonomy The autonomy of the device implies that it can function, even when it is no longer connected with any other participant. Concretely, this implies that the collaboration is not aborted when the device is temporarily disconnected. Instead it can continue making local changes and rest assured that these changes will percolate to the other participants upon reconnecting.

Natural Concurrency Since participants concurrently edit the joint document without necessarily being able to synchronise, different versions of the document may be in circulation. When the different versions are to be merged during synchronisation, editing conflicts may arise. For instance, if the collaborative editor is line-based, conflicts arise when one participant edits a line which the other participant had either edited or deleted. These application-specific conflicts will be signalled using the exception handling mechanism.

We consider the collaborative editor presented in this section to be an illustrative example which epitomises a class of ambient-oriented applications where state is replicated across different collaborating devices which are entitled to make independent updates. When these independent updates prove to be incompatible, an exception is raised. The next section outlines a set of criteria for an exception handling mechanism which can be applied in an ambient-oriented setting.

3 Ambient-Oriented Exception Handling

This section presents a collection of criteria that need to be exhibited by an ambient-oriented exception handling mechanism. Whereas some of the criteria described below can be observed in various distributed exception handling mechanisms, no single exception handling mechanism exhibits all of them.

Asynchronous Exception propagation Exception handling mechanisms define a *context* in which dynamically raised exceptions are to be handled. For instance, when an editor detects merge conflicts while it was asked to synchronise, the editor that requested the synchronisation needs to be brought back into the correct context for handling these conflicts. In an ambient-oriented setting, such contexts cannot be described using classic **try-catch** blocks since the processes making up an ambient-oriented application communicate using *non-blocking communication primitives*. This implies that the calling process may have left the context of its **try** block before the exception was propagated by the invoked process. The most basic criterion for an ambient-oriented exception handling mechanism is therefore that it provides an adequate mechanism for ensuring that exceptions raised by a concurrent process are caught in the correct context.

Concerted Exceptions Exception handling mechanisms typically allow specifying a single context for exception handling for an entire block of code consisting of several instructions. The combination of *non-blocking communication primitives* with block-level handling implies that all processes invoked by the block may concurrently raise exceptions. In the collaborative editor example, this situation may occur during the synchronisation which basically consists of broadcasting local changes to the other editors. Each editor may independently raise exceptions, which might have to be handled jointly. An ambient-oriented exception handling mechanism should therefore allow the programmer to examine all concurrently raised exceptions (*e.g.* to evaluate whether they are symptoms of a common problem) and to subsequently propagate a *concerted* exception [12] which best captures the particular combination of raised exceptions.

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. As a consequence, an exception raised by one process may violate these assumptions and therefore requires to be handled by all participating processes. The synchronisation in the collaborative editor application is an excellent example of this optimism. When synchronising, each editor will merge in all changes that do produce local conflicts. The design choice not to wait for confirmation from all participants, improves the resilience of the application to disconnection, but also implies that conflict handling is a collaborative activity, not only involving the editors which signalled a conflict, but also those who already performed the merge.

Loosely-coupled Exception Handling An ambient-oriented exception handling mechanism should guarantee the autonomy of the processes for which it handles exceptions. This implies that it may not rely on a centralised node to coordinate the exception handling. Furthermore, it needs to provide a mechanism to discover long-lasting disconnection, in order to prevent processes from waiting indefinitely for an unreachable communication partner. Long-lasting disconnection is signalled using an exception, albeit one raised by the environment rather than by the program.

4 Related Work

A diverse spectrum of approaches exists to handle exceptions in a distributed or concurrent setting. This section groups these existing approaches according to the level of granularity at which they operate and evaluates them with respect to the criteria for ambient-oriented exception handling mechanisms outlined in the previous section.

4.1 Message-level Handling

A single asynchronous message send is the finest level of granularity at which a distributed exception handling mechanism can operate. The flavour of an exception handling mechanism at this level depends largely on the underlying mechanism for communication. The non-blocking communication prescribed by the ambient-oriented programming paradigm can be achieved using two different communication mechanisms. Processes may communicate in a non-blocking way with one another directly using asynchronous messages or indirectly by reading and writing tuples in a shared distributed tuple space [19].

Asynchronous Messages Exception handling can be aligned with asynchronous message sending by passing a complaint address along with every message [10]. When an exception occurs, a predefined message (*e.g.* **handle**) is sent to the object denoted by that address, passing the exception object as a parameter. When futures are used to represent the results of asynchronous invocations, these futures need to be integrated into the exception handling mechanism. An example of such an integration is witnessed in the E language which provides a **when-catch** construct to specify how to handle results as well as exceptions raised from a particular asynchronous invocation [16].

Tuple Spaces Serugendo and Romanovsky [21] argue that exception propagation for distributed systems communication using tuple spaces is best handled using an external mechanism to ensure that each exception is correctly handled, rather than depending on the fact that the appropriate process will read the exception tuple and handle it. The CAMA system [11] therefore requires every tuple to be equipped with a reference to a *tuple space trap* to which exceptions are signalled. Such tuple space traps can transform the received exception and choose to propagate it to the "caller", a dedicated handler agent or an ensemble of (affected) agents.

Whereas these exception handling mechanisms may provide a fertile basis to develop an ambient-oriented exception handling mechanism, they lack support for funnelling concurrently raised to a single *concerted exception*, since they consider only one message send and thus one exception at a time.

4.2 Block-level Handling

Various distributed exception handling mechanisms offer a variation of the well-known **try-catch** construct, to bind a single exception handler to a sequence of asynchronous message sends. As we have indicated in section 3, the fact that the messages are sent asynchronously implies that different exceptions may be raised concurrently. Different mechanisms exist to reduce these concurrent exceptions to a single *concerted exception* :

ProActive aims to hide the induced concurrency and therefore only handles the first exception to be raised inside the **try** block [3]. The underlying assumption is that all asynchronous invocations are closely related (*e.g.* they depend on one another's results) such that the first exception is a good representative of the underlying error.

SaGE deals with exceptions raised from asynchronous invocations in a multi-agent system [22]. It requires handlers to provide a **concert** method that is invoked for every exception raised. Based on the current exception and the log of previous exceptions it can choose to either log the exception for future reference or to immediately propagate a concerted exception.

Arche aggregates *all* concurrently raised exceptions and feeds them to a *resolution function* that in its turn raises the concerted exception [12, 13]. The major 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 more complete information.

DOOCE [9] varies on the semantics of the `try-catch-finally` construct by allowing all exceptions raised by asynchronous invocations in a particular `try` block to be handled by the associated `catch` blocks. Exceptions that cannot be handled by these most closely nested `catch` blocks are aggregated and passed to the `finally` block after all asynchronous calls have returned.

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 messages. In other words, the mechanisms discussed above offer no provisions for *collaborative exception handling*, as prescribed in section 3. This implies that the techniques described above are only applicable when the different processes make no optimistic assumptions. In an ambient-oriented setting however, optimistic assumptions are often required to cope with the effects of volatile connections.

4.3 Collaboration-level Handling

Finally, some exception handling mechanisms allow structuring an application in 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-treaded Transactions (OMT transactions) structure a group of collaborating threads within the boundaries of a transaction [14]. The threads communicate using shared objects which maintain their own consistency to ensure that exceptions caused by one thread cannot influence any other thread. This form of communication is opted for to allow each thread to handle exceptions locally. This design decision implies that OMT transactions provide no support for *collaborative exception handling.*

Coordinated Atomic Actions (CA actions) are a well-established technique for describing a collaboration between different processes. The exception handling mechanism produces *concerted exceptions* using an exception graph [25], and raises this concerted exception in all participating processes. Whereas a lot of experience exists on how to distributed CA actions [20, 26], the model proves to be too rigid for our purposes since it requires every participant to exit the CA action with the same result. This implies that once a participant disconnects, the entire collaboration must be aborted.

The Guardian Model shuns the use of transaction-like mechanisms to structure applications for exception handling. Instead a process can explicitly manipulate its own context by pushing symbolic names on the context stack. Similarly, when raising an exception, the exception needs to be tagged with a handling context. All processes which are currently in this context will collaborate to handle this exception. However, since each process may change its context independently, the model needs to contact all processes whenever exceptions are raised. In order to ensure a timely response to exceptions, the guardian model relies on being able to contact all processes in a bounded time [17]. This requirement implies that the exception handling mechanism depends on the presence of *all* processes, which clearly conflicts with the *loosely-coupled exception handling* characteristic.

This section has provided a general overview of the different approaches to handle exceptions that were raised within the context of a collaboration of various processes : OMT transactions opt for a particular communication scheme that minimises the effect of a single exception, so that it may always be handled locally. CA actions on the other hand implement transaction-like guarantees to ensure that the effect of errors can be adequately handled collaboratively by its participants. Unfortunately this semantics is hard to reconcile with the characteristics of an ambient-oriented setting. Finally, the guardian model offers collaborative handling without imposing the use of a transaction-like structure, yet it proves to be impracticable precisely because of this lack of imposed structure.

4.4 Conclusion

Our overview of the existing distributed exception handling mechanisms has failed to identify a single exception handling approach that adheres to all of the criteria for an ambient-oriented exception handling mechanism. This shortcoming forms the main motivation for our work. On the other hand, the approaches discussed in this section provide a solid basis on which to base our own approach. For example, we have opted to employ an E-like mechanism to allow exceptions to be propagated from one actor to another one. This model was then extended with a language construct to protect a block of asynchronous invocations, whose exceptions are fed to an Arche-like resolution function. Finally, we have constructed a distributed mechanism for structuring collaborating actors, in a manner that is reminiscent of CA actions.

5 Ambient Conversations

This section describes ambient conversations, a first exception handling mechanism for ambient-oriented software. The ambient conversation model consists of an ensemble of four language constructs which directly correspond to the four criteria described in section 3. 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, the ambient conversation model provides for *loosely-coupled exception handling* through the introduction of the **due** construct.

5.1 Ambient-Oriented Programming in AmbientTalk

The ambient conversation model was realised as a reflective extension of the AmbientTalk language kernel, which was designed as a language laboratory to uncover new features for ambient-oriented programming languages. An in-depth description of the language is outside the scope of this paper, and can be found elsewhere [4]. For the sake of comprehending the exception handling mechanism explained below it suffices to know that AmbientTalk's concurrency model is based on actors [1] which communicate with one another using asynchronous message passing. Asynchronous messages are processed one by one by an actor's thread. This serial treatment of messages precludes actors from suffering from race conditions. AmbientTalk's actors are also the unit for distribution. Hence, an ambient-oriented application in AmbientTalk consists of a suite of actors that are possibly located on different devices and that send asynchronous messages to one another. Notice that AmbientTalk does not contain a notion of proxies. Once two AmbientTalk objects get to know each other through AmbientTalk's service discovery mechanism (which is outside the scope of this paper), they can transparently communicate with each other over a volatile connection. When connections are temporarily broken, messages sent are accumulated with the sender and are automatically flushed upon re-establishing that connection. AmbientTalk's asynchronous messages are denoted with the **#** operator. In what follows, we use a pseudo-syntax in order to avoid having to explain AmbientTalk's technical details. The low-level technical details of the reflective implementation of the constructs described in this paper can be found in a companion technical report [18].

By default, AmbientTalk's asynchronous messages do not return a result since such results would cause the sender to wait for them and this would be a source for blocking. Another problem with asynchronous message sends in the face of return values is how to deal with return values (and exceptions) in the proper calling context. Indeed, that context typically no longer exists when the result is returned because the sender of the message immediately proceeds after having sent the message. An existing solution for these problems is to deal with asynchronous return values using futures (or promises) [8, 15, 2]. The idea is that an asynchronous message send immediately returns a future; a placeholder for the return value which will be replaced by the real return value once it is computed. The future is then said to be *resolved* with the value. In [4] it is shown how AmbientTalk's default behaviour is reflectively extended for asynchronous messages to return such futures. This paper relies on this extension. Existing proposals for futures differ depending on how they solve the problem of messages being sent to futures which have not been resolved yet. The most common semantics employed is to *block* the sender of these messages until the future has been resolved. However, this violates the non-blocking and autonomy characteristics of ambient-oriented programming as explained in section 2.1. The paradigm dictates non-blocking send *and* receive operations, whereas waiting for the resolution of a future is clearly a blocking receive operation. Therefore, AmbientTalk adopts the non-blocking futures as proposed in the language E [16]. The idea is to return a new future when sending a message to an unresolved future. When the latter future is eventually resolved to a value, then the message is sent to that value and its result will resolve the new future. This is called future-pipelining and was independently termed promise-pipelining in E. Future pipelining allows one to "chain" asynchronous message sends, even though the intermediate results are not yet computed.

Very often, synchronisation is needed between the sender of an asynchronous message and the future pipelining. This occurs when the application logic dictates that a certain action is to be undertaken upon resolving a future. Surely, that action can be put as a method in the value that will resolve the future. However, this would clutter up the code significatly. Therefore, AmbientTalk and E feature a **when** language construct that takes three parameters: a future, a formal parameter name for the resolved value, and a code block. The idea is that the code block is *registered* with the future. Calling the **when** language 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. This is accomplished by binding the formal parameter name to the value to which the future was resolved. The point is to execute this block *when* the future that corresponds to its first argument has been resolved. As such, **when** allows one to send an asychronous 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 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 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 = actor # compute();
 when(fut) becomes(result) {
   display("second", result)})
 when(fut) becomes(result) {
   display("third", result)}))
  display("first")
}
```

5.2 Supporting Asynchronous Exception Propagation

Exceptions in AmbientTalk are represented as objects[1] that understand the **match** message which takes one argument. The argument is expected to be just another exception object. The message is used by the exception handling mechanisms to determine whether a raised exception matches an exception object specified by handlers as explained below. Programmers can make their own variants of exceptions as long as these implement that method.

An exception is raised by the **raise(e)** primitive, where **e** is expected to be an exception object. When this happens 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 pipeling semantics described above. As explained, when a message **m** is sent to a future $\mathbf{f}_1$, a new future $\mathbf{f}_2$ is returned that will be resolved by the result of sending **m** to the resolution of $\mathbf{f}_1$. However, when $\mathbf{f}_1$ is eventually ruined, $\mathbf{f}_2$ will be ruined by the same exception. A similar phenomenon exists in the E language where it is called broken promise contagion [16].

The future for a method depends only on the value of the last expression in the method's body. This implies that the method may return a value, irrespective of the fact that the futures of some of the asynchronous messages it sent may be ruined eventually. Section 5.3 presents an exception accumulation mechanism, which gathers *all* exceptions prior to resolving the future of a method. Before doing so, we first turn our attention to the way exceptions are caught along the future ruining pipeline.

Similar to the **when** construct described above, AmbientTalk features 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 with a value, and a block of code that might be executed when the future gets ruined by an exception. The construct looks as follows.

```
when(f) becomes(val) {
  when-block
} catch(Exception1) {
  catch-block1
} catch(Exception2) {
  catch-block2
  ...
}
```

[1] Apart from actors, AmbientTalk also features 'normal' objects that do not have any concurrency provisions. For the details we refer to [4].

The idea is that several **when-catch** constructs can register themselves with a future **f** and that every **when-catch** construct can list several catch clauses. Every k'th **when-catch** registered with a future **f** denotes a future $\mathbf{f'}_k$ in its turn. If **f** gets resolved with a value, all registered when-blocks will be executed. If **f** is ruined by an exception, all registered catch clauses are notified of the exception, such that they can determine whether one of their branches matches the raised exception (using the **match** message described above). 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 its turn. In the former case, the value is used to resolve $\mathbf{f'}_k$. In the latter case, the exception is used to ruin $\mathbf{f'}_k$.

5.3 Supporting Concerted Exceptions

By default, the result of a block of code is aligned with the value of the last expression in that block. Similarly, 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). This default semantics is however not always desirable. In the collaborative editor example, editors regularly broadcast their local changes, to synchronise with the other participants of the editing session. This synchronisation consists of sending independent **merge** messages to all editors participating in the writing session. In this case, the synchronisation is only to be considered successful if *none* of these participants has propagated an exception. It is precisely in such cases that a mechanism is required to funnel all possible concurrent exceptions and produce a single *concerted exception*.

AmbientTalk's exception handling mechanism 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 raise a *concerted exception*. Using the **group-resolve** construct, the synchronisation between different editors can be written as follows:

```
method synchroniseDocument(document) {
 group {
  for editorActor in editorActors {
   editorActor#merge(document);
  }
 } resolve( concurrentExceptions ) {
  // compute and raise a concerted exception based on concurrentExceptions
}}
```

Needless to say, 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.

5.4 Supporting Collaborative Exception Handling

The criteria for an ambient-oriented exception handling mechanism defined in section 3 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. The **synchroniseDocument** method described in the previous section, contains an example of such an optimistic operation in the collaborative editor example. The **merge** method which is invoked on all collaborating editors, allows each editor to merge the changes under the assumption that none of the other participants will have encountered conflicts. This assumption allows editors to autonomously merge changes without communicating with the other participants, thereby tolerating the temporary disconnection of some of the collaborating editors. However, when one of the editors *does* propagate back an exception, all editors need to be notified of this event, and be able to collaboratively handle the exception.

Our ambient conversation model achieves *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 ) {
 // Additional behaviour for 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. The participants are notified of such broadcasted exceptions by installing **when-catch** observers on the conversation actor. A **conversation** can thus be thought of as a special kind of future which can be 'ruined multiple times'. 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. Each participant is given a reference to its local replica using the **startConversation** method. A participant can broadcast an exception to all other participants in the conversation by invoking the **propagate** method on its local replica. Section 6 illustrates how conversations are used in the editor case in order to broadcast a merge exception to all participating editors.

5.5 Supporting Loosely-coupled Exception Handling

An exception handling mechanism can only be considered to be ambient-oriented, if it is a *loosely-coupled exception handling* mechanism. Concretely, the exception handling mechanism should not impose the use of any structure that may potentially harm the autonomy of a device, or make it vulnerable to the effects of volatile connections. As we have already discussed in the previous section, the conversation is replicated on each device hosting one of its participants precisely to avoid harming the autonomy of the participants in any way.

Although the **group-resolve** and **when-catch** constructs will never make an actor block, they cannot ensure that their futures will eventually be resolved or ruined. This may happen when the futures they observe are associated with messages sent to actors which have become permanently unreachable. As such these futures will never be resolved or ruined, and no registered when-block would ever trigger. Hence, support is needed to differentiate between temporary and long-lasting (presumably permanent) disconnections. When the disconnection lasts longer than a predefined timeout, the future ought to be ruined with an exception signalling the disappearance of the lost actor.

In previous work, we have already proposed the **due** language construct which solves the above problem by putting an expiration deadline (in milliseconds) on outgoing messages [4]. In this paper we provide a slightly modified **due** construct which employs futures and is integrated with the exception handling mechanism. The modified **due** construct consists of a timeout value (relative to the time at which a message is sent) and a code block which may contain multiple asynchronous message sends. The idea is to let the **due** denote a future which is to be resolved by the value of its code block. When these messages have not been delivered to their destination actor within the prescribed interval, the delivery of this message is cancelled, and its future is ruined by a **TimedOutMessage** exception. In the editor example of the following section, the **due** construct is used to exclude editors from the collaborative editing session after they have been disconnected for too long a period.

```
due( maxTimeOut ) {
  when(editorActor#merge(document)) becomes(result) {
    // merge performed successfully
  } catch ( TimedOutMessage ) {
    // remove editor from the collaborative editing session
  }
}
```

6 Implementation of the collaborative editor

This section illustrates how the ambient conversation model presented in the previous section can be used to realise the collaborative editor described in section 2.2. Such a collaborative editor can be embodied in the ambient-oriented programming paradigm as a suite of collaborating editor actors. Each editor is provided with its own copy of the document which it may edit locally. For simplicity, we assume the collaborative editor to be line-based with a public interface consisting of an **insert**, a **delete** and a **replace** method. These methods allow the editor to make local changes on its copy of the document. At regular intervals, the editor will synchronise its local document, with the version of its collaborators.

The synchronisation of one editor with its collaborators is achieved by sending each editor a **merge** message with its own document. The merge operation can encounter two kinds of conflicts while combining the changes in two documents: either one editor has replaced a line that was deleted by another participant or both of them have replaced the same line. These conflicts are reported by raising the **DeleteException** and **EditException** exceptions respectively.

In order for editors to collaborate, they need to establish a conversation listing all participants invited to join the editing process. This is performed by the **publish** method listed below, who receives an array of the editor participants and creates a conversation grouping them.

```
method publish(editorActors) {
 conversation(editorActors) {
      method synchroniseDocument(){
        group{
            for editorActor in editorActors {
              editorActor#merge(document);
      }
        } resolve( concurrentExceptions ) {
          //manage conflicts received from all the participant
          //and construct a concerted merge exception
          thisActor#propagate(aMergeException);
        }}
 }
}
```

As explained in section 5.4, the conversation actor sends a **startConversation** message to each participant passing along a replica of itself. The body of the conversation defines the protocol for broadcasting their changes to the different participant editor actors. Since the merge phase may report overlapping conflicts, the **group-resolve** construct is used here to handle and resolve them in a single concerted conflict. Note that while the **synchroniseDocument** method is clearly application-dependent, the propagation of exceptions to the participants will be triggered by the default **propagate** method implicitly defined by the conversation actor.

Each editor must also implement the **startConversation** method with the code that follows. As mentioned, upon reception of a **startConversation** message, all the participants will receive a replicated actor representing the collaborative editing session. Editors can then register a **when-catch** block on their local conversation replica to handle the exceptions raised during the synchronisation phase. If the synchronisation concludes successfully, the editor checkpoints its document.

```
method startConversation(conversationActor) {
 when(conversationActor) becomes(val) {
      //code to update the checkpoint
  } catch( DeleteConflict ){  // application-dependent code to manage the conflict
  } catch( EditConflict ){  // application-dependent code to manage the conflict
}}
```

The **due** construct can be used in the context of the collaborative editor example in order to detect long-lasting disconnection of participants and force them to leave the conversation. The code below shows how the **due** contstruct can be added to the **publish** method explained above. In the updated version, the **due** construct is used to stamp the **merge** messages sent to all the participant editors with a timeout value of **maxTimeOut** milliseconds. Each **merge** message is then wrapped in a **when-catch** block to handle the **TimedOutMessage** exception raised if the message is not delivered within **maxTimeOut** milliseconds. If a **TimedOutMessage** exception is signalled, the participant to which the timed out message was sent is removed from the conversation. Note that other exceptions raised by the **merge** method will be handled in the **resolve** block. Thus, handling the **TimedOutMessage** exception does not interfere with other raised exceptions in a **group-resolve** construct.

```
method publish(editorActors) {
 conversation(editorActors) {
      synchroniseDocument(document){
        group{
      for editorActor in editorActors {
             due(maxTimeOut){
               when( editorActor#merge(document) ) becomes (val) { ...
         } catch(TimedOutMessage){
                // the participant is removed from the conversation
               }
             }
         }
       } resolve( concurrentExceptions ) {
        //manage conflicts received from all the participants
        thisActor#propagate(aMergeException);
      }}
 }
}
```

The implementation of the collaborative editor application validates the language constructs designed for the ambient-oriented exception handling mechanism. Furthermore, the implementation also illustrates that it is relatively straightforward to combine the different constructs to build an ambient-oriented application which is resilient to both deliberately raised exceptions as well as exceptions raised due to lost participants.

7 Conclusion

This paper has focussed on the integration of exception handling into an ambient-oriented language called AmbientTalk. Having unravelled the hardware characteristics that fundamentally discriminate mobile networks from their stationary counterparts, we have identified four criteria for novel ambient-oriented exception handling mechanisms. We have then proposed the ambient conversation exception handling model as a suite of exception handling language features in which each language feature addresses one of the criteria. The essence of the ambient conversations exception handling model consists of the **when-catch** language construct which correctly propagates and handles exceptions resulting from an asynchronous message send between several actors that are possibly located on different devices linked by a volatile connection. Based on this language construct, the **group-resolve** mechanism was proposed to group several such asynchronous message sends when these occur in a block of code. Their concurrently raised exceptions can then be funneled into a single concerted exception. Third, the **conversation** exception broadcasting construct was described that allows one to specify that different actors are collaborating and to ensure that all available participants are involved in the exception handling process. Finally, the distribution properties of the proposed constructs were evaluated in the face of non-temporary disconnections. An additional fourth **due** mechanism was described to deal with this. The four constructs were validated by using them in the design of an ambient-oriented collaborative editor that allows several editors deployed on autonomous hardware to participate in a shared writing session. The exception handling constructs were used to make this editor resilient to distributed merge conflicts without relying on a shared infrastructure.

References

1. Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. D. Caromel and M. Rebuffel. Object based concurrency: Ten language features to achieve reuse. In R. Ege, M. Singh, and B. Meyer, editors, *Proceedings of TOOLS-USA'93, Santa Barbara, (CA), USA*, pages 205–214. Prentice-Hall, Englewood Cliffs (NJ), USA, 1993.
3. 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*, number 05-050 in Technical Reports - Laboratoire d'Informatique, de Robotique et Micro-Electronique de Montpellier, 2005.
4. Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, and Theo D'Hondt. Ambient-oriented programming in AmbientTalk. Submitted to

the 20th 6th European Conference on Object-Oriented Programming (ECOOP 2006), 2006.

5. Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.
6. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.
7. IST Advisory Group. Ambient intelligence: from vision to reality, September 2003.
8. Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
9. Shin ichi Tazuneki and Takaichi Yoshida. Concurrent exception handling in a distributed object-oriented computing environment. In *Seventh International Conference on Parallel and Distributed Systems Workshops (ICPADS'00 Workshops)*, 2000.
10. 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.
11. A. Iliasov and A. Romanovsky. Exception handling in coordination-based mobile environments. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, pages 341–350. IEEE Computer Society Press, 2005.
12. Valérie Issarny. An exception handling mechanism for parallel object-oriented programming: toward reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–40, 1993.
13. Valérie Issarny. Concurrent exception handling. In *Advances in Exception Handling Techniques (Lecture Notes in Computer Science)*, volume 2022, pages 111–127. Springer-Verlag, 2000.
14. Jörg Kienzle, Alfred Ströhmeier, and Alexander Romanovsky. Open multithreaded transactions: Keeping threads and exceptions under control. In *Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, page pp. 197, 2001.
15. B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.
16. M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In R. De Nicola and D. Sangiorgi, editors, *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.
17. Robert Miller and Anand Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE Trans. Software Eng.*, 30(12):1008–1022, 2004.
18. Stijn Mostinckx, Jessie Dedecker, Elisa Gonzalez Boix, Tom Van Cutsem, and Wolfgang De Meuter. Ambient-oriented exception handling in ambienttalk. Technical report, Vrije Universiteit Brussel, 2006.
19. A. Murphy, G. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.

20. A. Romanovsky and A. F. Zorzo. On distribution of coordinated atomic actions. *SIGOPS Oper. Syst. Rev.*, 31(4):63–71, 1997.
21. Giovanna Di Marzo Serugendo and Alexander Romanovsky. Using exception handling for fault-tolerance in mobile coordination-based environments. In *ECOOP Workshop on Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms*, 2003.
22. F. Souchon, C. Dony, C. Urtado, and S. Vauttier. Improving exception handling in multi-agent systems. In *Advances in Software Engineering for Multi-Agent Systems*. Springer-Verlag, 2003.
23. Robert Tolksdorf and Kai Knubben. Programming Distributed Systems with the Delegation-based Object-oriented Language dSelf. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 927–931. ACM Press, 2002.
24. M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.
25. 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.
26. 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.