Reconciling Generic Functions with Actors

Generic Function-driven Object Coordination in Mobile Computing

Jorge Vallejos   Pascal Costanza   Tom Van Cutsem   Wolfgang De Meuter   Theo D'Hondt

Programming Technology Lab – Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{jvallejo, pascal.costanza, tvcutsem, wdemeuter, tjdhondt}@vub.ac.be

Abstract

Event-driven programming is recognised as an appropriate paradigm for coping with dynamically reconfigurable environments such as mobile computing. This paradigm – essentially described in the *actor* model – has been properly aligned with *message-passing* method invocation systems in object-oriented programming. However, the combination with the *generic function*-based system still remains an issue, due to the mismatch between the actor's message-sending semantics and generic function invocations. Existing approaches circumvent this problem by (re)introducing a "send" operation which considerably diminishes the power of generic functions. In this paper, we describe an object-oriented programming model for distribution and concurrency, called *Lambic*, that reconciles generic functions and event-based programming. In this model, events are represented as asynchronous function invocations which are sequentially processed by event loops dispatching to the appropriate generic functions. We validate this approach by implementing a number of programming language abstractions for event-driven service coordination.

Keywords Generic functions, Actors, Mobile computing, Common Lisp, Lambic

1. Introduction

Event-driven programming has increasingly gained popularity among emerging distributed computing paradigms such as mobile computing (Van Cutsem et al. 2007b). In this programming style, software services communicate with each other by means of events. In the object-oriented literature, the essence of event-driven programming has been studied through the functional formalism of concurrency known as the actor model (Agha 1986), in which events are modelled as messages asynchronously exchanged among software entities (originally actors, but also objects and components in later extensions). In the past, actor-based concurrency and distribution have been successfully combined with object-oriented programming by aligning the message-based communication model of actors with the *message-passing* method invocation model of objects initially proposed in Smalltalk (Varela and Agha 2001; Miller et al. 2005; Van Cutsem et al. 2007b). However, the combination with the *generic function*-based method invocation style – like the one featured in CLOS – remains still an issue. In the message-passing model, objects are directly *addressed* by sending messages to them which conforms the message-sending semantics of actors. However, in the generic function-based model only functions can be directly addressed. Objects can only be *referenced* as arguments in function invocations.

It appears that one is forced to abandon the generic function metaphor if one wants to express event-driven concurrency and distribution. Existing Common Lisp language extensions have acknowledged this problem by introducing message-passing semantics into the language, e.g. a single-dispatching "send" operation (Harbo 2008; Gerrits 2005; Gorrie 2002; Hotz and Trowe 1999). However, if the non-distributed part of a service is written using generic functions, the overall service is then forced to combine the two different paradigms, which is far from trivial. Futhermore, such a combination seriously diminishes the benefits of generic functions. As Peter Seibel states in (Seibel 2005), by separating methods from classes, generic functions turn method dispatching inside out compared to message-passing. As such, programmers can specialise methods on multiple parameters (multiple dispatch), work with multiple inheritance in a much more manageable way, and use declarative constructs to control the method combination (auxiliary methods).

In this work, we present an actor-based object-oriented programming model that reconciles event-driven programming with generic functions for concurrency and distribution. The main idea of this model is to represent event notifications as asynchronous generic function invocations which are sequentially processed by the actors dispatching to the appropriate generic functions.

We validate our work by implementing an event-driven object-oriented coordination model for mobile computing defined in (Van Cutsem et al. 2007b), which was originally developed for message-passing systems. We demonstrate that in our implementation objects discover, communicate and react upon changes in their network environment without requiring single-dispatch operations.

2. Integrating Actors and Generic Functions

In this section, we discuss the benefits of event-driven concurrent and distributed programming by describing its manifestation in the AmbientTalk programming language (Van Cutsem et al. 2007b). The design decisions implemented in this language have significantly influenced the requirements for our model based on events and generic functions, as we explain in Section 2.3. We further review other approaches that combine event-driven programming and object-orientation at the end of this section.

2.1 Actors

The actor model (Agha 1986) is a concurrency model that represents concurrent activities as separate agents or *actors*. Actors have been conceived as a functional model, but extensions have been formulated that combine the actor paradigm with imperative programming. When we refer to actors throughout the rest of this paper, we consider them as concurrent processes that share no state and communicate strictly by means of asynchronous message passing. Each actor has a mailbox, which is a queue that buffers the messages already received but not yet processed by the actor.

Actors are of interest to us because they provide a clean abstraction of a distributed system. In a distributed system, processes generally have no shared state and communicate by means of network messages. Furthermore, even on a local machine, the fact that actors cannot synchronously access one another's state is useful because it prevents race conditions on that data.

2.2 Combining Objects and Actors

In early actor languages (e.g. Lieberman's ACT-1 language (Lieberman 1987)), all values are represented as actors. While this enables a flexible and uniform programming model, it also makes local sequential, non-distributed computing more complicated than strictly necessary. Recently, the E (Miller et al. 2005) and AmbientTalk (Van Cutsem et al. 2007b) programming languages have introduced an execution model that allows objects and actors to gracefully co-exist. In these languages, actors are not represented as objects but rather as containers of regular objects. Each object is *owned* by an actor. An actor can own multiple objects, but each object is owned by exactly one actor.

Actors define boundaries of concurrent execution around groups of objects. Two objects owned by the same actor can communicate synchronously, by means of traditional message passing. However, objects may refer to objects owned by other actors. Object references that span different actor boundaries are named *far references* and only allow asynchronous access to the referenced object. Any message sent to a receiver object via a far reference is enqueued in the mailbox of the actor that owns the receiver object and processed by the owner itself. Actors are event loops: they take messages one by one (i.e. sequentially) from their mailbox and dispatch them to the receiver object by invoking its appropriate method. Figure 1 depicts actors and objects in AmbientTalk.

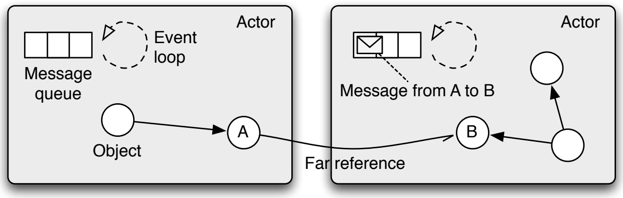

Figure 1. Actors in AmbientTalk

E and AmbientTalk combine the actor primitives to send and receive asynchronous messages with object-oriented message passing. For example, E and AmbientTalk do not feature an explicit `receive` statement to receive messages from remote objects. Rather, message reception is represented simply as method invocation. An object can thus accept any messages for which it defines a corresponding method.

E and AmbientTalk allow asynchronous message sends to return values by means of *futures* (Halstead, Jr. 1985), also known as *promises* (Liskov and Shrira 1988). A future is a placeholder for the return value of an asynchronous message send. Once the return value is computed, it replaces the future object; the future is then said to be resolved with the value.

To make the discussion more concrete, consider the following example. Assume `messenger` represents a far reference to a remote object that represents an instant messenger application (further explained in the next section). The following code shows how to query this instant messenger for its user's username:

```
def f := messenger<-getName();
when: f becomes: { |val| display(val) }
```

The `<-` operator denotes an asynchronous send of the message `getName` to the remote `messenger` object. This operation returns a future `f`. In AmbientTalk, an object may react to a future becoming resolved by registering an observer (a closure) that will be called with the resolved value (`val`), when the future has become resolved. To be able to respond to a `getName` message, it suffices for the messenger object to define a getName method as follows:

```
def createMessenger(name) {
  object: {
    def getName() { name }
  }
}
```

Note that the return value of the getName method is used to resolve the future that was created as a result of the messenger<-getName() message send.

2.3 Actors and Generic Functions United

The combination of event-driven and object-oriented programming implemented in AmbientTalk strongly relies on the message-passing method invocation model – events are aligned with messages. Using generic functions, message passing could be easily simulated by specialising the methods of generic functions on a single "receiver" parameter (single dispatch), as in NetCLOS (Hotz and Trowe 1999) or CL-MUPROC (Harbo 2008), for example. However, as we previously explained, this would severely diminish the benefits of the generic function-based style. To overcome this problem we introduce an event-driven object-oriented programming model that reconciles actors with generic functions. We have implemented this model as an extension to CLOS, called *Lambic*. In the remainder of this paper we refer to the concurrency model and to *Lambic* interchangeably.

Lambic is built around three key features: *actors as object containers*, *asynchronous method invocations* and *asynchronous return values*. This section describes these features, but first, we briefly introduce an example used in this description to illustrate the features.

The Instant Messenger Application Figure 2 shows part of the implementation in *Lambic* of an instant messenger application for mobile ad hoc networks, originally presented in the AmbientTalk literature (Van Cutsem et al. 2008). This application features two interfaces (represented by the methods specialised on the im-local-facade and im-remote-facade classes) which separate the local operations of the instant messenger from those that are remotely accessible. In this implementation, the local interface defines the talk method that enables the messenger's user to send messages to their "buddies" (users recorded in the buddies hash table), and the remote interface defines the receive method that enables the user to receive text messages. Both interfaces are subclasses of the distributable-object class which is the root for all the classes in *Lambic*. The last function defines an instant messenger application. We have omitted the coordination details required for mobile ad hoc networks. Section 4 shows the complete version of this function.

Actors as Object Containers

Lambic features a variation of the AmbientTalk's communicating event loops model. As in AmbientTalk, actors are object containers defining boundaries of concurrent execution for the objects. Event notifications are modelled as asynchronous generic function invocations which are sequentially processed by the actor's event loop, dispatching to the appropriate generic functions. Events are then handled by the corresponding methods in the generic function. Figure 3 depicts the actors in *Lambic*.

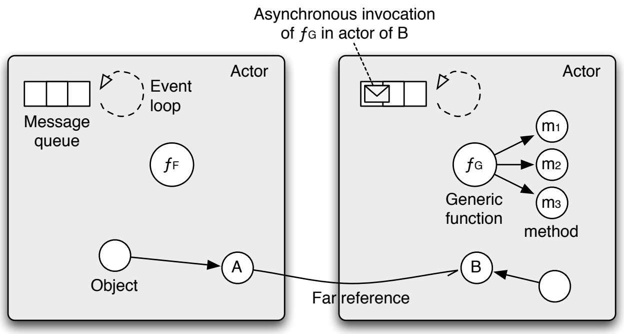

Figure 3. Actors in *Lambic*

The following expression shows the way to create actors in *Lambic* (using the defactor construct):

(**defactor** *name object**) ⇒ *actor*

For example:

```
(defactor "im-actor"
          local-interface remote-interface)
```

An actor is defined by indicating the name and the objects that it will own. As result, a reference to the object representing the actor is returned[1]. The example above shows the actor used in Figure 2 to represent the instant messenger application. This actor, named "im-actor", contains the local-interface and remote-interface objects, which are instances of the im-local-facade and im-remote-facade classes respectively.

By default, *Lambic* provides an actor that represents the (virtual) machine. This actor contains all the objects that are not owned by any other actor.

Asynchronous Generic Function Invocations

In our model, standard (synchronous) generic function invocations are only allowed if they occur within the actor that owns all the objects used as arguments. Inter-actor computations are possible by means of *asynchronous* generic function invocations. A generic function is asynchronously invoked by designating an actor as the responsible for its processing. This results in scheduling the function invocation in

[1] This object is also considered to be owned by the actor

```
; TEXT-MESSAGE
(defstruct text-message from content)

; LOCAL-FACADE
(defclass im-local-facade (distributable-object)
  ((username :initarg :username :accessor im-local-facade-username)
   (buddies :initarg :buddies :initform (make-hash-table :test #'equal)
            :accessor im-local-facade-buddies)))

(defmethod talk (local-facade to text)
  (let ((buddy (gethash to (im-local-facade-buddies local-facade))))
    (if buddy
      (let ((message (make-text-message :from (im-local-facade-username local-facade)
                                        :content text)))
        (in-actor-of buddy (receive buddy message) :without-future)
        (format t "Message to ~D: ~D~%" to text))
      (format t "Unknown buddy: ~D~%" to))))

; REMOTE-FACADE
(defclass im-remote-facade (distributable-object)
  ((username :initarg :username :accessor username)))

(defmethod receive (remote-facade message)
  (format t "Message from ~D: ~D~%" (text-message-from message)
          (text-message-content message)))

; FUNCTION TO CREATE A INSTANT MESSENGER ACTOR
(defun create-instant-messenger (username)
  (let ((local-facade (make-instance 'im-local-facade :username username))
        (remote-facade (make-instance 'im-remote-facade :username username)))
    (defactor "im-actor" local-facade remote-facade)
    ...
    local-facade))
```

Figure 2. Definition of an instant messenger application in *Lambic*

the event queue of the actor. The following expression illustrates an asynchronous function invocation:

(**in-actor-of** *object function-invocation*) ⇒ *future*

For example:

```
(in-actor-of messenger (username messenger))
```

In *Lambic*, neither actors nor objects can receive messages directly. In order to select the actor that should process an asynchronous function invocation, a programmer has to supply a reference to an object contained in the targeted actor. Similar to AmbientTalk, objects in *Lambic* are referenced from outside their actor's boundaries by means of *far references*.[2] This model enables objects to be serialised as far references by extending the default behaviour of objects. Thus, an asynchronous function invocation can be read as *"process this function invocation in the actor of this object"*. The example above corresponds to the translation of the AmbientTalk message that queries for the name of the instant messenger's user, described in Section 2.2. In this asynchronous function invocation, `messenger` is a far reference to the remote interface object of a remote messenger application. By passing this far reference as the first argument to the `in-actor-of` form, we ensure that the invocation to the `username` accessor method (defined in the `im-remote-facade` class) is processed by the actor that contains the remote interface object.

Note that this invocation corresponds to a "send" operation at the actor level which we claimed incompatible with generic functions. However, in our case this operation does not conflict with the generic function machinery as methods are only specialised on objects, not on actors. The only role of actors is to prevent concurrent invocations using the same set of objects.

[2] In *Lambic*, the serialisation semantics of objects (to enable objects to be distributed as far references) are specified in the `distributable-object` class.

Asynchronous Return Values

By default, an asynchronous generic function invocation returns a future as result. A future is an object created at the actor from where the function is invoked, that acts as placeholder for the result of the invocation. Observers can be defined to express actions that depend on the result by means of the following expression:

(**when-resolved** *future lambda*) ⇒ *future*

For example:

```
(when-resolved
  (in-actor-of messenger (username messenger))
  (lambda (name)
    (format t "Buddy name: ~D~%" name)))
```

In this expression, the `lambda` represents the action to be executed once the future is resolved with the result. `when-resolved` does not block the actor's event loop (to display the name of the messenger's user in the example).

Asynchronous generic function invocations may also indicate the future object that should be resolved with the result (by using the `:with-future` keyword argument). Conversely, one-way invocations (without result) can be expressed by including the keyword `:without-future`. In such a case, the invocation does not return any value. The latter is the case of the asynchronous invocation inside the body of the `talk` method of the `im-local-facade` class in Figure 2:

```
(in-actor-of buddy (receive buddy message)
             :without-future)
```

In this case, we use the `:without-future` keyword to indicate that no result is expected for the invocation of the `receive` method of a remote messenger application.

To the best of our knowledge, no other Common Lisp library features each of the above properties. We further discuss the related work in the next subsection.

2.4 Related Work

In this section, we compare *Lambic* with existing Common Lisp libraries for distribution and concurrency. Back in the 80's and beginnings of 90's we find a considerable number of Lisp dialects that supported concurrent (also called parallel) programming (Halstead, Jr. 1985; Yuen and Wong 1990; Clamen et al. 1990). However, we do not further review those approaches as they do not work with generic functions.

NetCLOS

NetCLOS (Hotz and Trowe 1999) is an actor-based extension to CLOS for parallel computing inspired by the concurrent object-oriented language ABCL/1 (Yonezawa et al. 1986). NetCLOS exhibits a double-layered object model based on actors (called active objects) and objects (called passive objects). As in *Lambic*, NetCLOS introduces message passing only to determine the place (the actor) where a function is evaluated. However, for this to happen the methods should be specialised on actors. In NetCLOS, a programmer has to deal with actors and objects at the same level, which is not always trivial. For instance, the programmer must decide which parts of the program behaviour should be represented as actors and the ones that should be modelled as objects. In our model, we avoid this problem by allowing the methods to be specialised only on objects and their classes.

NetCLOS provides three ways to invoke functions by means of the *past*, *now* and *future*-messages. The kind of invocation accepted by a generic function is indicated in its definition (using the :past, :now and :future keywords). In *Lambic*, the decision on how a generic function is invoked can be made only at invocation time (synchronously or asynchronously in our case). Furthermore, asynchronous function invocations are only possible by means of the `in-actor-of` language abstraction.

Erlang in Common Lisp

Erlang is an actor-based functional programming language specially designed for distributed concurrent computing (Armstrong et al. 1996). It extends the actor model providing support for reliable message sending among actors (called *processes*) over the network. Upon reception, messages are handled by means of a pattern matching operation (described in `receive` blocks). At present, there are a number of libraries that introduce the Erlang's concurrency model to Common Lisp, e.g. CL-MUPROC (Harbo 2008), Distel (Gorrie 2002) and Erlisp (Gerrits 2005) [3]. However, in those libraries we find very little integration with the generic function-based object-oriented programming style provided by CLOS. As result of this, the programmers are forced to reason differently about the distributed and/or concurrent part of the program (dealing with message sending and pattern matching) and the local and non-concurrent part of the program (dealing with generic function invocations and method definitions). While in *Lambic* there is also a difference between local and remote function invocations, none of these invocations require extra language support to be handled (like `receive` blocks).

Clojure

Clojure (Hickey 2008) is a Lisp dialect that runs on the JVM with special focus on non-distributed concurrent programming. This language does not support object orientation. However, it does provide a combination of multimethods and agents which is comparable to our model. Multimethods differs from generic functions in that they generalise the type-based dispatching to one based on any arbitrary function on the arguments. Agents are like actors in that they support asynchronous sharing of mutable state between threads. However, agents do not have an event loop and no blocking `receive` (they rely on a thread pool). Functions – called actions – are dispatched on agents by means of a send oper-

[3] Although Erlisp was never finished, we refer to the informal discussion of its ideas

ation which always return immediately. Dispatch functions can be associated with multimethods which produces a similar result to asynchronous function invocations in *Lambic*. Agents can synchronise actions by means of an *await* operation which causes the current thread to block until the actions are performed (unlike observers on futures, which do not block the actor's thread).

Distributed Generic Functions

In (Queinnec 1997), Queinnec presents an extension to generic functions for distributed computing. The main concern in this work is to avoid inconsistent modifications of generic functions shared by different locations. Implementations of generic functions in Lisp and Scheme typically rely on side effects when defining and adding new methods to existing generic functions, which may lead to race conditions in distributed settings. This is avoided by adopting a purely functional approach, declaring the generic functions as immutable. Other than that, this model assumes generic functions to have only one discriminating argument (the receiver) and an explicit send operation. Furthermore, this approach is not based on the actor model.

2.5 Summary

In summary, *Lambic* aligns actors and generic functions as follows:

- Actors define boundaries of concurrent execution for the objects. Methods are specialised on objects and their classes, not on actors.
- Inter-actor computations are realised by means of asynchronous generic function invocations. This kind of invocation differs from standard generic function invocations in that it has to specify the actor that should evaluate the function.
- By default, the result of an asynchronous generic function invocation (if any) is also asynchronously returned to the actor from which the function is invoked (the future's location).

3. Generic Function-driven Object Coordination in Mobile Computing

We illustrate the benefits of *Lambic* by implementing a programming language model for software service coordination in mobile computing environments. This implementation is based on the coordination model of AmbientTalk which is specially designed for a particular kind of mobile environment known as mobile ad hoc networks (acronym MANETs) (Dedecker et al. 2006). In this section, we outline the issues of MANETs for service coordination identified in the AmbientTalk literature, describe the programming abstractions found in that language, and show their implementation in *Lambic*.

3.1 Coordination in Mobile Ad Hoc Networks

MANETs exhibit two hardware phenomena that fundamentally differentiate these networks from traditional, fixed computer networks (Van Cutsem et al. 2007b). The first phenomenon, known as *volatile connections*, is related to the fact that connections between different mobile devices may be easily interrupted when users move about with their devices. In many cases the disconnection is temporary which means that the devices may meet again and require their connection to be re-established. The second phenomenon present in MANETs is their very little or non-existent fixed infrastructure (known as *zero infrastructure*). In such networks, services become dynamically available according to the connection state of the devices that contain them. A service that relies on other services in its environment to perform its task needs to be aware of the availability of such devices.

The above phenomena entails a number of requirements for the coordination of services in MANETs which (Van Cutsem et al. 2007a) characterises as follows:

Decentralised service discovery Any application that relies on other devices to perform its task should be able to discover its dynamic network environment. Yet this discovery should not be centralised in a particular machine (i.e. a fixed server), as there is no certainty that the connection with such machine will be permanently available. A decentralised service discovery protocol needs to be introduced to enable the services to autonomously act upon the availability and unavailability of nearby services.

Decoupled communication In MANETs, the communication between services should be independent of the volatile connectivity of their devices. This means that the services do not necessarily need to be online at the same time to communicate with each other (time decoupling). Similarly, the potentially extensive periods of disconnection during a communication imply that synchronisation between different parties should be performed without blocking their control flow, i.e. without suspending their thread of control (synchronisation decoupling).

Services should also be able to communicate without knowing each other's address or location beforehand (space decoupling). This enables the services to better adapt to changes in their physical environment as the conceptually same service may be provided by several instances hosted by different devices.

Connection-independent failure handling In MANETs, services should be able to perform failure handling independent of any network failure. The reason for this is that disconnections can be transient and as such the services may want to resume computation upon reconnection. Treating disconnections as a normal mode of operation is an optimistic form of partial failure handling (and also higher level than physical network failures).

3.2 Event-driven Object Coordination in AmbientTalk

AmbientTalk satisfies the coordination criteria presented above by modelling service discovery, communication and failure handling as events. In addition to the asynchronous object communication mechanism based on event loops, far references and futures (described in Section 2.2), that language provides the two extra characteristics for coping with service coordination in MANETs: support for discovery and failure handling.

AmbientTalk employs a publish/subscribe service discovery protocol. Objects are published by means of types (space decoupling) while a subscription takes a form of the registration of a discovery event handler on a type tag, which is triggered whenever an object exported under that tag has become available in the ad hoc network. As result of the discovery, a far reference to the exported object is received. This protocol is fully decentralised, no servers or other infrastructure are required.

In AmbientTalk, far references are resilient to partial failures (including network partitions). When a network or machine failure occurs, a far reference becomes disconnected. A disconnected far reference buffers all messages sent to it (time decoupling). When the failure is restored at a later point in time (e.g. a network partition is healed), the far reference flushes all accumulated messages to the remote object in the same order as they were originally sent. Hence messages sent to far references are never lost, regardless of the internal connection state of the reference. To enable such failure handling, AmbientTalk introduces two failure event handlers (triggered upon disconnection and reconnection). The closures corresponding to the handlers are asynchronously applied whenever the interpreter detects the disconnection (respectively reconnection) of the object referred to by the far reference.

3.3 Event-driven Object Coordination in *Lambic*

In order to fully preserve the semantics of *Lambic* (described in Section 2.3), a coordination model for MANETs based on actors and generic functions should represent coordination events – for discovery, communication and failure handling – as generic function invocations. Such invocation should be resilient to partial failures. In what follows we describe the language abstractions implemented in *Lambic* that support these requirements.

Discovery Abstractions

Lambic features a decentralised publish/subscribe service discovery protocol similar to AmbientTalk. Objects can be exported and imported by means of type tags denoting the services they provide. The functions that enable these actions obey the following patterns:

(**export-service** *object service-description*) ⇒ *object*
(**import-service** *service-description*) ⇒ *channel*

The `import-service` function returns a particular class of object called *channel*. A channel is a proxy to the requested service which waits for an object providing the service (a *far reference* to that object) to become available.[4] Once this happens, the channel creates a binding to the object. The functionality of the imported object can be accessed by means of asynchronous function invocations. Such invocations are processed at the actor of the imported object by passing the channel as the actor designator (the first argument of the `in-actor-of` function). A channel can be used in an asynchronous function invocation even if the service object it represents is not yet discovered. In such a case, the "unbound" channel buffers the invocations until the object is discovered. After the discovery, the channel flushes all the stored invocations.

Lambic also provides a way to register discovery event handlers. This is done by using the following functions:

(**when-discovered** *channel lambda*) ⇒ *channel*
(**whenever-discovered** *service-description lambda*)
⇒ *nil*

Whereas the `when-discovered` function enables the programmer to define actions to be processed after an imported service is discovered (i.e. after the channel representing that service is bound to a far reference), the `whenever-discovered` function enables the programmers to define actions to be processed whenever an object providing the requested service becomes available. In both cases, a channel to the discovered service is passed as argument to the lambda.

Failure Handling Abstractions

Lambic provides the two failure event handlers for disconnection and reconnection proposed by AmbientTalk.

(**when-disconnected** *channel lambda*) ⇒ *channel*
(**when-reconnected** *channel lambda*) ⇒ *channel*

These two functions enable programmers to define actions to be executed when an imported service becomes disconnected and reconnected respectively. As in the discovery functions, the lambda of the disconnection and reconnection function receives a channel as argument.

Channels also provide support for failures. In case of disconnection, the channel that represents the service becomes unbound which means that it buffers the asynchronous function invocations that indicate this channel as the actor designator. Once the connection to the service is restored, the channel sends the accumulated invocations.

[4] Channels are similar to a language abstraction in AmbientTalk called *ambient references* (Van Cutsem et al. 2007a).

3.4 Discussion

In summary, our generic function-based coordination model accomplishes the coordination criteria described in this section as follows:

- Services are discovered using a decentralised publish/subscribe protocol. Objects can be exported and discovered by means of type tags denoting the service they provide. No address or location is required (space decoupling).
- Coordination events (for discovery, communication and failure handling) are all represented as function invocations. Event handlers can be defined for each of these events.
- Communication events correspond to asynchronous generic function invocations (synchronisation decoupling).
- Asynchronous generic function invocations are resilient to partial failures (by using channels). When an actor that is supposed to process a remote invocation becomes disconnected, the invocation is buffered until the connection is restored (time decoupling).

Note that none of the features above requires message-passing semantics. The message-based actor interaction is cleanly separated from the programming level, replaced by a number of coordination functions. Methods do not need to provide a receiver argument. Programmers do not send messages to objects but only invoke generic functions and eventually indicate the actor that should process it (by giving a reference to an object owned by such actor).

4. *Lambic* in Action

We illustrate the use of the above coordination model by completing the implementation of the instant messenger application introduced in Section 2.3.

4.1 An Instant Messenger for MANETs

Figure 4 shows the complete definition of the function `create-instant-messenger`. This function performs a number of actions. First, it creates the local and remote interfaces by instantiating the `im-local-facade` and `im-remote-facade` classes. Second, it defines an actor that contains such instances (although this is not required if we assume that there is only one instant messenger per location, in which case the default actor is sufficient to represent the instant messenger's boundaries of concurrent execution). Third, this function publishes the remote interface. The service description used in this implementation, contained in the `instant-messenger` local variable, follows the convention of a discovery service protocol called Zero Configuration Networking (IETF 1999). Finally, this function defines a number of event handlers for discovery, disconnection and reconnection of other instant messengers.

The following code illustrates how to create instant messenger and send a message.

```
(defvar *bob-im*
        (create-instant-messenger "Bob"))
(talk *bob-im* "Alice" "Hi, Alice")
```

4.2 Coordination Abstractions in Action

In this example, the coordination abstractions are used as follows.

Discovering Instant Messengers

Instant messengers discover each other by means of the `whenever-discovered` form (inside of the `create-instant-messenger` function). Whenever an instant messenger is encountered, the discovery handler (the lambda argument of the `whenever-discovered` function) notifies the discovery (displaying a message), registers the discovered instance (a channel) in the `buddies` hash map of the `im-local-facade` object of the instant messenger, and defines the failure handlers. The discovery handler is invoked only if the discovered instance has not been previously registered and if the discovered and discovering messengers are not the same instance. Both conditions are checked by means of the name of the users of the applications.

Communicating Instant Messengers

Most of the implementation of this application requires synchronous generic function invocations. Asynchronous generic function invocations are only used for communication between instant messengers. The `talk` method of the local interface of the instant messenger asynchronously invokes the `receive` function (as explained in Section 2.3). An additional asynchronous generic function invocation is required to get the user name of a discovered instant messenger inside of the discovery handler previously presented. Similar to the previous case, the `username` accessor function is evaluated by the actor that owns the remote `im-remote-facade` object which is connected to the channel represented by `messenger`. However, this invocation does require a result. For this reason we put the invocation inside of a `when-resolved` form.

Handling Volatile Connectivity of Instant Messengers

In this implementation, text messages can be sent to the discovered instant messengers (by means of the `talk` function) even if they are disconnected. This is possible thanks to the use of channels which store the messages until the instant messenger they point at becomes available again.

The disconnection and reconnection of the discovered instant messengers are notified (displaying a message) by means of handlers specified in the `when-disconnected` and `when-reconnected` functions respectively.

```
(defun create-instant-messenger (username)
  (let ((local-facade (make-instance 'im-local-facade :username username))
        (remote-facade (make-instance 'im-remote-facade :username username))
        (instant-messenger-type "_p2pchat._tcp."))
    (defactor "im-actor" local-facade remote-facade)
    (export-service remote-facade instant-messenger-type)
    (whenever-discovered instant-messenger-type
      (lambda (messenger)
        (when-resolved (in-actor-of messenger (username messenger))
          (lambda (buddy-name)
            (unless (or (gethash buddy-name (im-local-facade-buddies local-facade))
                        (equal buddy-name (im-local-facade-username local-facade)))
              (setf (gethash buddy-name buddies) messenger)
              (format t "~D added to ~D's buddy list~%" buddy-name username)
              (when-disconnected messenger
                (lambda (messenger)
                  (format t "~D offline!~%" buddy-name)))
              (when-reconnected messenger
                (lambda (messenger)
                  (format t "~D online!~%" buddy-name)))))))))
    local-facade))
```

Figure 4. Function to create an instant messenger in *Lambic*

4.3 Evaluation

The implementation of the instant messenger in *Lambic* described in this section is comparable to the original implementation in AmbientTalk in terms of functionality and number of lines (excluding the parts of the implementation in AmbientTalk that were not covered in our work, like GUI and leasing abstractions). Furthermore, we achieve the goal of using a programming style based on generic functions.

5. Implementation

In this section, we discuss the main details of the implementation of *Lambic* in Common Lisp together with the AmbientTalk-like coordination model for MANETs. We use LispWorks®(Enterprise Edition 5.1.1) as our development platform.

5.1 Implementing Event-driven Concurrency

In *Lambic*, event-driven concurrency is realised by the functions defined for the `actor` class, to *send*, *receive* and *process* messages. The `send-message` function, invoked in the expansion of the `in-actor-of` macro, has two methods which handle local and remote asynchronous function invocations (specialising a `receiver` argument in the `distributable-object` and `far-reference` classes respectively). In both cases the asynchronous invocations are converted into messages. Local asynchronous invocations are handled by invoking the `receive-message` function which puts the message in the event queue of the actor. Remote asynchronous invocations are forwarded to a *communication* actor that transmits the invocations over the network. Both `send` methods immediately return a future – an instance of a `future` class.

The event loop of an actor in *Lambic* is represented by a process which is provided by the Multi-Processing library of LispWorks®. The actor's event queue corresponds to the mailbox of a such process. The `defactor` macro creates an instance of the `actor` class and initiates a process which continuously waits for a message in its mailbox. Upon message reception, the process calls the `process-message` function of the actor which converts the message into a generic function invocation, invokes the function, and sends the result to the actor that contains the future attached to the message (if any).

In the definition of the distribution and concurrency model implemented by *Lambic* (described in Section 2.3), we state that function invocations that spawn the actors' boundaries must be asynchronous. However, thus far this is not enforced at the implementation level. Objects owned by actors can still be used in synchronous function invocations by any process in the system. We then leave the responsibility to the programmers to follow the principles defined by our model.

Asynchronous return values are achieved by means of the functions `when-resolved` and `resolve-with-result` defined for the `future` class. `when-resolved` (directly used in programs, as described in Section 2.3) receives a future and a lambda that is executed if the `value` field of the future is bound, or stored in the `continuations` list otherwise. The `resolve-with-result` function is asynchronously invoked when the actor that processes the asynchronous invocation for which the future serves as the return value's placeholder, produces the result.

5.2 The Coordination Model

The coordination model for MANETs implemented in *Lambic* (presented in Section 3.3) relies on a Common Lisp library called CL-ZEROCONF (Wiseman 2005). This library provides an interface to the implementation of the Zero Configuration Networking protocol by Apple Inc., Bonjour (Apple Inc. 2008). The language abstractions for discovery and failure handling are built on top of callback functions provided by CL-ZERCONF to discover, resolve and remove services.

In *Lambic*, messages are reliably transmitted over the network using channels (instances of the `channel` class) which abstract the socket-based communication proposed by the Communication library of LispWorks®. Channels work in a similar manner to futures. A channel represents the connection to a remote actor and as such it transmits messages only if such actor is available in the network, and stores them otherwise until the actor is (re)connected.

6. Conclusion and Future Work

In this work, we focus on actor-based distribution and concurrency to cope with the dynamicity of mobile computing environments. We identify the mismatch between the actor's message-sending semantics and the generic function-based method invocation model. We then propose an object-oriented programming language model that reconciles actors and objects. In our model, actors define boundaries of concurrent execution for the objects; methods are specialised on objects (not on actors); inter-actor computations are realised by means of asynchronous generic function invocations; and the results of functions evaluations are asynchronously returned to the actor from which the functions are invoked. Therefore, the actor's message-sending semantics are explicitly separated from the programming level, which enables the programmers to invoke functions instead of sending direct messages to objects and to define methods without receiver arguments.

We implement this model as an extension to Common Lisp called *Lambic*. We use *Lambic* to validate our work by implementing a number of functions for service coordination in mobile ad hoc networks (discovery, communication and failure handling) based on the coordination model of AmbientTalk. We illustrate the use of such functions developing an instant messenger application for such networks.

Currently, we are investigating different extensions of our model that benefit from the combination of actors and generic functions. We are exploring how to integrate our model with generic function-based approaches for expressing context-dependent behavioural adaptations (like ContextL (Costanza and Hirschfeld 2005) or Filtered Dispatch (Costanza et al. 2008)), a property that is also considered essential for services running on dynamically reconfigurable environments like MANETs. Such an integration would enable programmers to model *distributed* context-dependent adaptations, which is one of our ultimate goals.

We are also investigating alternatives to bring the semantics of synchronous and asynchronous generic function invocations closer to each other, which would facilitate the evolution of programs, similarly to approaches that unify synchronous and asynchronous message-based object communication (Caromel 1993).

Acknowledgments

This work has been supported by the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and the MoVES project of the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy. The authors would also like to thank Sebastian Gonzalez, Engineer Bainomugisha, Yves Vandriessche and Charlotte Herzeel for the valuable discussions during this work.

References

Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. ISBN 0-262-01092-5.

Apple Inc. Networking Bonjour Protocol, 2008. URL `http://developer.apple.com/networking/bonjour`.

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

Denis Caromel. Towards a Method of Object-oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, 1993.

Stewart M. Clamen, Linda D. Leibengood, Scott Nettles, and Jeannette M. Wing. Reliable Distributed Computing with Avalon/Common Lisp. *Computer Languages, 1990., International Conference on*, pages 169–179, Mar 1990. doi: 10.1109/ICCL.1990.63772.

Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-Oriented Programming - An overview of ContextL. In *Dynamic Languages Symposium*, 2005.

Pascal Costanza, Charlotte Herzeel, Jorge Vallejos, and Theo D'Hondt. Filtered Dispatch. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2.

Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, 2006.

Dirk Gerrits. Erlisp, Common Lisp Library, 2005. URL `http://common-lisp.net/project/erlisp`.

Luke Gorrie. Distel: Distributed emacs lisp (for erlang). In *Erlang User Conference*, 2002.

Robert H. Halstead, Jr. MULTILISP: a Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/4472.4478.

Klaus Harbo. CL-MUPROC: Erlang-inspired Multiprocessing in Common Lisp, 2008. URL `http://common-lisp.net/project/cl-muproc`.

Rich Hickey. Clojure, 2008. URL `http://clojure.org`.

Lothar Hotz and Michael Trowe. NetCLOS - Parallel Programming in Common Lisp. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

IETF. Zero Configuration Networking (Zeroconf), 1999. URL `http://www.zeroconf.org`.

Henry Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

Barbara H. Liskov and Liuba 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. ISBN 0-89791-269-1.

Mark S. Miller, Dean E. Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.

Christian Queinnec. Distributed Generic Functions. In *In Proc. 1997 France-Japan Workshop on Object-Based Parallel and Distributed Computing*, 1997.

Peter Seibel. *Practical Common Lisp*. 2005. ISBN 1-59059-239-5 (hardcover).

Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Object-Oriented Coordination in Mobile Ad Hoc Networks. In *COORDINATION*, 2007a.

Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. AmbientTalk: Object-Oriented Event-driven Programming in Mobile Ad hoc Networks. In *XXVI International Conference of the Chilean Computer Science Society (SCCC)*. IEEE Computer Society, 2007b.

Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jorge Vallejos, and Jessie Dedecker. Ambient-Oriented Programming website, 2008. URL `http://prog.vub.ac.be/amop`.

Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/583960.583964.

John Wiseman. CL-ZEROCONF Library, 2005. URL `http://projects.heavymeta.org/rendezvous`.

Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented Concurrent Programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986. ISBN 0-89791-204-7.

Chung-Kwong Yuen and Weng-Fai Wong. A Self Interpreter for BaLinda Lisp. *SIGPLAN Not.*, 25(7):39–58, 1990. ISSN 0362-1340.