A Modular Mixin-based Implementation of Ambient References

Tom Van Cutsem*
Programming Technology Laboratory
Vrije Universiteit Brussel

May 15, 2006

Abstract

This document describes the implementation of Ambient References as reflective extensions to the AmbientTalk programming language. The implementation is based on a composition of three kinds of mixin objects which interact according to an object-oriented protocol based on abstract methods. The implementation of all kinds of ambient references is explained. Furthermore, the extensions made to the discovery mechanism of AmbientTalk to support ambient references are detailed. Finally, the usage and implementation of sustained message sends, snapshots and multifutures which may be used in tandem with ambient references are described.

This document assumes the reader understands the AmbientTalk programming language [DVM+06]. and has a thorough understanding of the design and the taxonomy of ambient references, which are described elsewhere [VDM+06].

1 Introduction

Ambient references have been implemented in the actor-based ambient-oriented programming language AmbientTalk. This minimal kernel features a metaobject protocol for its actors, allowing one e.g. to change their default message receipt and message sending behaviour. It is this metaobject protocol which allows AmbientTalk to be extended from within itself with novel language features. Ambient references have been implemented reflectively on top of the more low-level event-based discovery system of the kernel. Furthermore, because the low-level discovery system of the language does not allow matching based on subtyping or filter queries, these extensions have been added in the form of a metacircular discovery mechanism, The design of this discovery mechanism is explained first. Subsequently, it is shown how ambient references are implemented as actors whose default semantics has been altered using the MOP.

The paper is structured according to the following sections:

Services and Service Discovery describes the extended service discovery mechanism based on *service types*, offering a more rich mechanism for two AmbientTalk actors to get acquainted.

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

Ambient Reference Mixins constitutes the bulk of the article. It explains the core implementation on which all kinds of ambient references are based and the different kinds of mixin extensions which may be mixed into the core implementation to acquire a specific kind of ambient reference.

Sustained Message Sends introduces support for sending sustained messages to ambient omnireferences. This allows clients to send messages to potential principal actors which may be encountered in the future.

Snapshots explains the usage and implementation of the `snapshot` operation (or meta-message), which allows for the fixation and enumeration of the volatile principal sets denoted by some of the ambient references.

Multifutures describes ambient references' support for futures and the `whenEach` and `whenAll` language constructs.

Ambient Reference Observers describes two useful language constructs (`when_found` and `when_lost`) that allow for the registration of observers with ambient references in order to monitor their binding behaviour.

2 Services and Service Discovery

The default service discovery mechanism of AmbientTalk is insufficient to implement the scope of binding of ambient references. First, the discovery mechanism only allows discovery based on simple tags or strings. This rules out more complex matching based on a subtype relationship between service types. Second, the discovery mechanism has no notion of property objects or filter queries to quickly advertise or filter based on service properties.

In order to address these shortcomings, an extended service discovery mechanism has been implemented in AmbientTalk itself. The implementation is based on a metacircular definition of the native discovery mechanism itself. However, rather than allowing for the provision and querying of simple tags, this extended implementation allows service actors to provide service types and property objects, or to require service types accompanied by filter queries. The implementation of the extended discovery mechanism can be found in the file `discovery.at`.

2.1 Service Types

Service discovery using ambient references is based on *service types*. Their implementation can be found in the file `servicetypes.at`. The following code shows how to define a hierarchy of service types:

```
servicetype(DeviceT, Service);
servicetype(PrintOptionT,DeviceT);
servicetype(PrinterT, PrintOptionT);
servicetype(FaxT, PrintOptionT);
servicetype(ColorT, Service);
servicetype(ColorPrinterT, ColorT, PrinterT);
```

Note that the `servicetype` function automatically recognizes its first parameter as a symbol name and will declare it in the scope in which the function is called. The

service type `Service` is the root service type. Service types are always declared to be a subtype of one or more other service types. The service type `Generic` is the least element of the service type hierarchy.

Service type matching is based on the subtyping relation $\preceq$. A provided service type P matches a required service type R if and only if $P \preceq R$, i.e. the provided type may be more specialized than the type asked for, but may never be more general. Note that the $\preceq$ relation is reflexive, so $S \preceq S$. For any service type S, it follows that $S \preceq$ `Service` and `Generic` $\preceq S$.

2.2 Service Actors

The `ServiceMixin` mixin whose implementation can be found in the file `service.at` defines an interface with the new service discovery mechanism. Moreover, it defines a new function `service`, which – when invoked with a behaviour object – returns an actor with the `ServiceMixin` mixed in. A service actor has access to the methods `provide` and `require`. `provide` takes a service type and optionally a property object as argument and makes `thisActor` advertise the given service type. The method `require` takes a service type and optionally a one-argument lambda acting as a filter query on the property object and makes `thisActor` require the given service type. An actor which uses `require` *must* implement the following callback methods:

```
serviceDiscovered(resolution) :: { };
serviceLost(resolution) :: { };
```

They are invoked by the discovery manager when a matching service is discovered. The `resolution` argument is an object with four public fields: `provider`, containing the mail address of the providing actor, `providedType`, the service type advertised by the provider, `requiredType`, the service type requested by the actor and `properties`, referring to the property object describing the service actor's "service attributes" (which is an empty object if none was provided by the service). Consider the following example of a printer service and a corresponding client.

service

```
PrinterBehaviour :: futuresMixin(root.object({
  print(msg) :: { display("printing ",msg,eoln) };
  init() :: {
    props : object({ dpi :: 400 });
    provide(ColorPrinterT, props)
  }
}));
makePrinter() :: service(PrinterBehaviour);
```

client

```
PrinterClientBehaviour :: root.object({
  printer : void;
  print(msg) :: {
    if(is_void(printer),
      "could not print message",
      { printer#print(msg);
        "message sent to printer" })
  };
  init() :: { require(PrinterT) };
  serviceDiscovered(resolution) :: {
    printer := resolution.provider;
    display("got a printer of type ",resolution.providedType.getName(),eoln)
  };
  serviceLost(resolution) :: {
```

```
    display("lost printer!",eoln)
  }
});
```

3 Ambient Reference Mixins

The file `AmbientRefMixins.at` defines the `AmbientRefsMixin` language mixin. This mixin augments an actor with ambient reference constructors and support for multifutures and sustained message sends. A small usage example follows:

```
actor(AmbientRefsMixin(root.object({
  printer : void;
  init() :: {
    printer := ambientFragileUni(PrinterT, lambda(p) -> { p.dpi > 300 })
  };
  ...
})));
```

More details about the `AmbientRefsMixin` are explained later on. We first turn our attention to the implementation of ambient references. Ambient references are implemented as actors. The behaviour of an ambient reference actor is represented by an `AmbientReference` object. This core object contains the behaviour which is common to *all* kinds of ambient references. It also defines the behaviour for an ambient reference's scope of binding. The core object uses the discovery mechanism explained above to translate generic discovery and MOP events into more high-level events which are relevant to ambient references. The different strategies to deal with these events depend on the elasticity and the cardinality of an ambient reference and have been factored out into separate mixins. Mixins are composed via delegation. In order to create a complete ambient reference actor behaviour, first an `AmbientReference` core object is created. Next, one of three cardinality mixin objects is created whose parent is the core object. Finally, one of three elasticity mixins is created with as its parent the cardinality mixin. This elasticity mixin object together with its parent and grandparent describe the complete behaviour of one kind of ambient reference and can be used as the behaviour of an ambient reference actor.

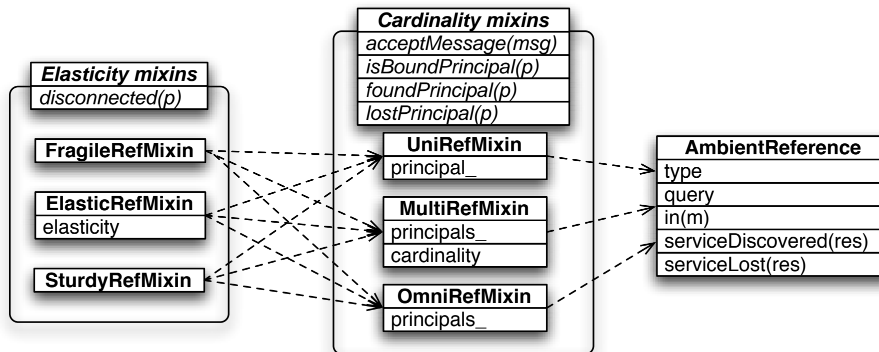

Figure 1: Ambient References as delegating mixin objects

Figure 1 illustrates the implementation of ambient references as the composition of one of three elasticity mixins, one of three cardinality mixins and an AmbientReference object. The figure shows objects whose names correspond to the constructor method that creates them. For example, the object named FragileRefMixin is created by invoking the constructor method makeFragileRefMixin(parent) where parent is a cardinality mixin object. The dotted arrows represent potential delegation. Each object has at most one parent object; the elasticity mixins are shown delegating to all cardinality mixins to emphasize that either one of them can act as their parent. This flexibility of delegation enables the modular construction of all kinds of ambient references. The slanted, abstract methods shown above the two mixin groups are methods implemented by all of the mixins in the group. Their implementation is required by the AmbientReference object. In the following sections, the implementation of the core object and the two groups of mixin objects is described.

3.1 Core Ambient Reference Object

The core AmbientReference object which is part of any ambient reference actor's behaviour is created by calling the following constructor method:

```
AmbientReference
makeAmbientRefCore(type, query) :: extend_with_args(root, lambda(type,query) -> {
  // auxiliary methods
  isMetaMessage(msg) :: { ... };
  forwardCarbonCopy(msg, to) :: { ... };
  forwardMessage(msg, to) :: { ... }
  // MOP methods
  init() :: {
    .init();
    inbox.addAddObserver(thisActor()#in);
    require(type, query)
  };
  in(msg) :: {
    if(!isMetaMessage(msg),
      { this().acceptMessage(msg) })
  };
  serviceDiscovered(resolution) :: {
    discoveredService : resolution.provider;
    joinedbox.add(discoveredService);
    if(!this().isBoundPrincipal(discoveredService),
      { this().foundPrincipal(discoveredService) })
  };
  serviceLost(resolution) :: {
    joinedbox.delete(resolution.provider);
    if(this().isBoundPrincipal(resolution.provider), {
      this().disconnected(resolution.provider)
    })
  }
}, [type, query]);
```

The purpose of the above object is to translate generic MOP and discovery events into meaningful ambient reference events. This is achieved by the mailbox observer methods, registered with their respective mailboxes in the init method, and via the discovery callback methods. After registering its mailbox observers, the ambient reference declares that it is interested in receiving notifications from the kernel pertaining to services providing the required service type (via require(type, query)). Note that arguments to the constructor function are translated into instance variables of the core object via extend_with_args.

Some method calls in the bodies of the mailbox observer and callback methods remain unimplemented by the core object itself. Such methods denote abstract methods

whose implementation should be provided by an appropriate mixin object. Each of these methods denotes a separate aspect of the ambient reference protocol. Each aspect is detailed below.

In the `in` inbox observer, the ambient reference first checks whether the incoming message is marked as a *metamessage*. Metamessages are those messages sent to the ambient reference actor itself, rather than client messages sent *via* the reference to the principal(s). An example metamessage is `snapshot` which should be handled by the ambient reference itself, rather than sent to the principal. The implementation of snapshot is deferred to section 5. The message acceptance semantics depends on the cardinality of the ambient reference, which is why the responsibility of dealing with message acceptance is delegated to a cardinality mixin's `acceptMessage` method. Figure 2 illustrates the message acceptance protocol.

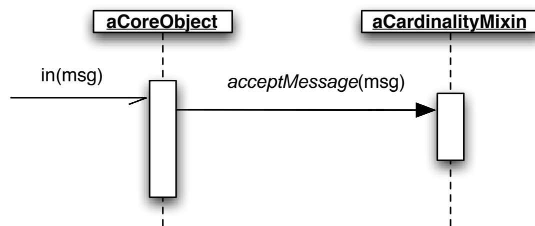

Figure 2: Message Acceptance Protocol

The `serviceDiscovered` callback method is invoked whenever a matching service actor joins the network. The scope of binding of an ambient reference is implicitly implemented via the discovery mechanism: the discovery manager invoking the callback implicitly guarantees that the discovered actor's provided type satisfies the required type and that its property object has passed the given filter query. The new service is a potential principal. Before propagating this event to the cardinality mixin which is responsible for binding the principal, the core object first checks whether the newly discovered actor isn't already bound. An ambient reference may never bind with the same actor more than once simultaneously. Because the act of binding a new principal is again dependent on the cardinality of the ambient reference, its implementation is factored out into a cardinality mixin's `foundPrincipal` method. Figure 3 illustrates the principal binding protocol.

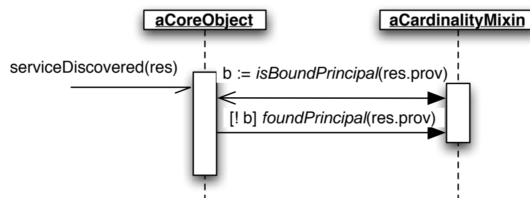

Figure 3: Binding Protocol

The `serviceLost` discovery callback is invoked whenever a joined service actor disappears from the network. The ambient reference only has to undertake action

upon such an event when the lost service is a bound principal. Whether a service is a principal or not can only be determined by the cardinality mixin of the reference. If the lost service turns out to be a bound principal, the ambient reference delegates the responsibility of dealing with this event to an elasticity mixin by invoking the abstract `disconnected` method. Figure 4 illustrates the disconnection resilience protocol.

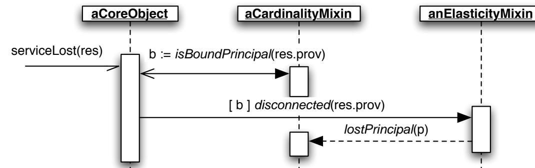

Figure 4: Resilience Protocol

Note that the protocol allows a mixin implementor to make the following assumptions:

- The `msg` parameter of `acceptMessage` never refers to a meta message.
- The principal argument to `foundPrincipal` is guaranteed to be a currently unbound candidate principal.
- The principal argument to `disconnected` is guaranteed to be a currently bound principal. For all ambient references described here, it also holds that `lostPrincipal`'s principal argument is a currently bound principal.

The following sections describe the cardinality and elasticity mixin objects, which provide implementations for the abstract methods of the ambient reference object.

3.2 Cardinality Mixins

A cardinality mixin is responsible for managing the principal(s) of an ambient reference. This includes providing and managing the principal set and handling the forwarding of received messages to the principal(s). A cardinality mixin must implement the following methods:

```
AbstractCardinalityMixin() :: object({
  acceptMessage(msg) :: { error("abstract method: acceptMessage") };
  isBoundPrincipal(p) :: { error("abstract method: isBoundPrincipal") };
  foundPrincipal(p) :: { error("abstract method: addPrincipal") };
  lostPrincipal(p) :: { error("abstract method: lostPrincipal") }
});
```

The three different kinds of cardinality mixins are described below.

3.2.1 Ambient Unireferences

A unireference's principal set consists of at most one element. Therefore, it is implemented simply as a variable named `principal` whose value determines the state of the ambient reference. If `principal` contains `void`, the reference is unbound. If it contains the mail address of a principal, the reference is bound. This variable is

toggled between `void` and a principal mail address in the `foundPrincipal` and `lostPrincipal` methods.

```
UniRefMixin
makeUniRefMixin(parentObject) :: extend(parentObject, {
  principal_ : void;
  acceptMessage(msg) :: {
    if(!is_void(principal_),
      { forwardMessage(msg, principal_) })
  };
  isBoundPrincipal(p) :: { principal_ ~ p };
  foundPrincipal(newPrincipal) :: {
    if(is_void(principal_), {
      principal_ := newPrincipal;
      inbox.asVector().iterate(lambda(msg) -> {
        forwardMessage(msg, newPrincipal)
      })
    })
  };
  lostPrincipal(p) :: {
    principal_ := void;
    outbox.asVector().iterate(lambda(msg) -> {
      if(msg.getTarget() ~ p, {
        inbox.addPrior(msg); //add to front to preserve ordering!
        outbox.delete(msg)
      })
    })
  }
});
```

The message forwarding behaviour of a unireference is described in the `acceptMessage` method. If the reference is bound, the incoming message is rerouted to the bound principal. If the reference is unbound, the message is kept in the inbox. The `foundPrincipal` method determines how the reference reacts upon the availability of a new candidate service provider. The appropriate behaviour depends on the state of the unireference: if it is unbound the new service becomes the principal, if it is bound the new service is disregarded, but is kept in the joinedbox for possible later use. Upon binding to a new principal, the unireference flushes its inbox and forwards it all messages accumulated while it was unbound. A message forwarded using `forwardMessage` is removed from the inbox.

The `lostPrincipal` method can be regarded as the inverse of the `foundPrincipal` method. When the currently bound principal disconnects, the reference reverts to unbound status. It may occur that messages forwarded to the disconnected principal were not successfully transmitted. Therefore, all untransmitted messages destined for the old principal are retracted from the outbox and added to the front of the inbox again, such that they are retransmitted upon rebinding to a principal in the future. Note that the `lostPrincipal` method does not check whether the lost principal `p` equals the current `principal`. This check is already performed via the `isBoundPrincipal` invocation in the `serviceLost` method of the core ambient reference object.

3.2.2 Ambient Multireferences

A `MultiRefMixin` is parameterized by its cardinality and is used to create an ambient multireference. This reference's principals are stored in an array whose size equals its cardinality. A multireference's cardinality denotes the maximum number of principals the reference can bind to simultaneously.

```
MultiRefMixin
makeMultiRefMixin(parentObject, cardinality) ::
 extend_with_args(parentObject, lambda(cardinality) -> {
```

```
  principals_[cardinality] : void;
    firstFreeIdx : 1;
    // auxiliary method, take a lambda(slot,index) and returns nothing
    forEachSlot(block) : { ... };
    // auxiliary method, takes a lambda(slot), returns slot idx or cardinality+1 if not found
    findFirstSlot(block) : { ... };
    bound_slot(slot) : { !is_void(slot) };
    acceptMessage(msg) :: {
      if(!is_number(msg.getTarget()), {
        forEachSlot(lambda(slot,idx) -> {
          if(bound_slot(slot), {
            forwardCarbonCopy(msg, slot)
          }, {
            save: msg.copy();
            save.setTarget(idx);
            inbox.addSilent(save)
          })
        });
        inbox.delete(msg)
      })
    };
    isBoundPrincipal(p) :: { findFirstSlot(lambda(slot) -> { slot ~ p }) <= cardinality };
    foundPrincipal(p) :: {
      if (firstFreeIdx <= cardinality, {
        principals_[firstFreeIdx] := p;
        inbox.asVector().iterate(lambda(msg) -> {
          dest: msg.getTarget();
          if(is_number(dest) & dest = firstFreeIdx, {
            forwardMessage(msg, p)
          })
        });
        firstFreeIdx := findFirstSlot(lambda(slot) -> { !bound_slot(slot) })
      })
    };
    lostPrincipal(p) :: {
      lostPrincipalIdx: findFirstSlot(lambda(slot) -> { slot ~ p });
      principals_[lostPrincipalIdx] := void;
      outbox.asVector().iterate(lambda(msg) -> {
        if(msg.getTarget() ~ p, {
          outbox.delete(msg);
          msg.setTarget(lostPrincipalIdx);
          inbox.addPrior(msg)
        })
      });
      firstFreeIdx := minimum(firstFreeIdx, lostPrincipalIdx)
    }
}, [ cardinality ]);
```

Upon receiving a message, the `acceptMessage` method first checks whether the message is addressed to a slot index (its recipient is a number). If this is the case, the message is ignored, as it is a message to be saved by the ambient multireference for later use, as explained below. Upon receiving a regular message, the message is duplicated for each principal slot. When a principal slot is bound (i.e. does not contain `void`), the message is immediately forwarded to the bound principal. More precisely, a copy of the message is forwarded. When a principal slot is unbound, (a copy of) the message remains in the inbox with its destination set to the index of the unbound slot, to be forwarded later on.

When a new candidate principal is found, the multireference first checks whether it still has a free unbound slot available to bind the new reference. If the multireference is entirely bound, the candidate principal remains stored in the joinedbox for possible later use. A principal is always bound to the smallest slot index available. Whenever a principal is bound to a slot, all messages whose destination is its slot index are removed from the inbox and sent to the new principal.

When a bound principal disjoins, its slot is unbound and all untransmitted messages sent to the principal are retracted from the outbox. Such retracted messages are added

in the correct order to the inbox and their destination is changed to point to the slot index rather than the lost principal.

3.2.3 Ambient Omnireferences

An omnireference stores its principal set in an extensible vector object. There is no limit to the number of principals an omnireference can simultaneously bind to. An omnireference mixin requires the `afterMixin` defined in the file `after.at` – which contains the definition of the `after` language construct – to be mixed in. Its use is detailed in the support for sustained message sends in section 4.

OmniRefMixin

```
makeOmniRefMixin(parentObject) :: extend(afterMixin(parentObject), {
  principals_ : vector.new();
  acceptMessage(msg) :: {
    principals_.iterate(lambda(principal) -> {
      forwardCarbonCopy(msg, principal)
    });
    // support for sustained message sends, explained below
    ...
  };
  isBoundPrincipal(p) :: { principals_.contains(p) };
  foundPrincipal(newPrincipal) :: {
    principals_.add(newPrincipal);
    inbox.asVector().iterate(lambda(msg) -> {
      forwardCarbonCopy(msg, newPrincipal)
    })
  };
  lostPrincipal(p) :: {
    principals_.remove(p);
    outbox.asVector().iterate(lambda(msg) -> {
      if(msg.getTarget() ~ p, {
        outbox.delete(msg)
      })
    })
  }
});
```

Upon receiving a message, the message is duplicated and sent to each principal currently in the principal vector. The omnireference's support for sustained message sends is deferred to section 4. Regular ("ephemeral") messages are simply deleted from the inbox immediately.

Upon notification of a new candidate principal, the candidate is added to the principal set and receives a copy of all messages currently kept in the inbox. This mechanism enables sustained messages waiting in the inbox to be received by principals which bind *after* the message has been received. When a bound principal is considered lost, it is removed from the principal set and any untransmitted messages forwarded to it are deleted.

3.3 Elasticity Mixins

An elasticity mixin specifies a policy on how to deal with disconnections of principals. Whenever a disconnection occurs, an elasticity mixin's `disconnected` method is invoked from within the core object. An elasticity mixin can decide whether the principal must actually be removed from the principal set or not. An elasticity mixin must implement the following method:

```
AbstractElasticityMixin() :: object({
  disconnected(p) :: { error("abstract method: disconnected") }
});
```

This method is invoked by the core ambient reference object whenever a bound principal disjoins. It is the responsibility of the elasticity mixin to invoke `lostPrincipal` if the ambient reference should effectively remove the bond with the disconnected principal. The different kinds of elasticity mixins are described below. Fragile and elastic ambient references share similar behaviour, which is factored out into a `BreakableRefMixin`. Before describing this mixin, we first describe the fragile and elastic ambient reference mixins.

3.3.1 Fragile Ambient References

The code for fragile ambient references is shown below:

FragileRefMixin

```
makeFragileRefMixin(parentObject) ::
  extend(makeBreakableRefMixin(parentObject), {
    disconnected(principal) :: { erase(principal) }
  });
```

The `disconnected` method of a fragile ambient reference immediately detaches a disconnected principal by invoking its parent object's `erase` method. This method's implementation is discussed below. What is important is that fragile ambient references align network disconnections with the immediate loss of a principal.

3.3.2 Elastic Ambient References

Elastic ambient references are parameterized with an elongation period (denoting their elasticity) which is a timeout period, in milliseconds, describing how long a principal may stay disconnected before becoming unbound. The code for elastic ambient references is shown below:

ElasticRefMixin

```
makeElasticRefMixin(parentObject, elasticity) ::
 extend_with_args(afterMixin(makeBreakableRefMixin(parentObject)),
  lambda(elasticity) -> {
    disconnectedPrincipals_ : smallmap.new(); // a map from principal to an expiration ID
    expirationId_ : 0;
    // method overridden from core object
    serviceDiscovered(resolution) :: {
      if(this().isBoundPrincipal(resolution.provider), {
        disconnectedPrincipals_.delete(resolution.provider)
      });
      .serviceDiscovered(resolution)
  };
  disconnected(principal) :: {
    expirationId_ := expirationId_ + 1;
    markedExpirationId : expirationId_;
    disconnectedPrincipals_.put(principal, markedExpirationId);
    after(elasticity, lambda() -> {
      currentExpirationId: disconnectedPrincipals_.get(principal);
      if(!is_void(currentExpirationId) & (currentExpirationId = markedExpirationId), {
        disconnectedPrincipals_.delete(principal);
        erase(principal)
      })
    });
    void
  }
}, [ elasticity ]);
```

Elastic ambient references require the `afterMixin` to be mixed in, and as mentioned above "inherit" from a `BreakableRef` mixin. To keep track of which currently bound principal is currently disconnected, an elastic reference has a map `disconnectedPrincipals` mapping principal mail addresses to expiration IDs. These expiration IDs uniquely identify particular disconnections. The reason for introducing them is explained below.

When an elastic reference's `disconnected` method is invoked, the principal that has gone astray is marked as disconnected by placing it in the disconnected map, but the principal is *not yet* removed from the principal set. Instead, the elastic reference waits for its specified elongation period (denoted by `elasticity`) to time out and then checks whether the disconnected principal is still in the principal set. The `after` language construct provided by the `afterMixin` is used to accomplish this non-blocking wait. The construct takes a timeout period and a zero-argument lambda and executes the closure asynchronously after the timeout period has elapsed. If the principal is still disconnected after the elongation period has elapsed, the principal is detached from the ambient reference using the breakable reference's `erase` method, making an elastic reference exhibit the same behaviour as a fragile reference.

Note that, in order to check whether the principal is still disconnected, the expiration ID plays a crucial role. More particularly, it is checked whether the principal stored in the disconnected map is still linked to the same expiration ID (`currentExpirationId`) as the one it was marked with at the start of the timeout period (`markedExpirationId`). This check is necessary to abolish race conditions which may occur due to a principal connecting and disconnecting multiple times during a single timeout period. For example, a principal may disconnect from an elastic reference with a 2-minute elongation period. If the principal rejoins after 1 minute but leaves again 30 seconds later, there are now two `after` blocks scheduled for execution. The first to trigger is the one scheduled when the principal originally disconnected. However, because the principal already reconnected after 1.5 minutes, the principal should not yet be considered as lost. To deal with this race condition, each disconnected event giving rise to a timeout period is uniquely identified by an ID such that it can be checked that the delayed code indeed deals with the latest disconnection or whether it has become obsolete.

It remains to be explained how a disconnected principal can be considered as reconnected. An elastic reference overrides the core object's `serviceDiscovered` method to check if a discovered service is a bound principal. If this is the case, that principal is removed from the disconnected principals set such that any delayed code dealing with its disconnection will be silently ignored. The overridden `serviceDiscovered` method always performs a super-send to execute the default binding behaviour. Note that, if the discovered service is a bound principal and thus possibly removed from the disconnected principal map, it will not give rise to a call to `foundPrincipal` because the original `serviceDiscovered` method of the core object does not invoke this method when discovering an already bound principal.

3.3.3 Sturdy Ambient References

The implementation of a sturdy reference mixin is simple. Its `disconnected` method does nothing, thereby silently disregarding the disconnection event of the principal. Any messages sent to the principal while it is disconnected keep on being forwarded and await transmission in the outbox. The ambient reference implementation falls back on the AmbientTalk kernel's ability to properly flush untransmitted messages in the outbox when it detects that the receiver has reconnected. Because the core ambient

reference object's `serviceDiscovered` method checks whether a candidate principal is already bound, the principal is not inserted into the principal set multiple times when it rejoins. The implementation of sturdy references is shown below:

SturdyRefMixin

```
makeSturdyRefMixin(parentObject) :: extend(parentObject, {
  disconnected(principal) :: { void }
});
```

3.3.4 Breakable Ambient References

Fragile and elastic ambient references factor out their common behaviour in the `erase` method of a breakable reference mixin. The implementation of the latter mixin is shown below:

BreakableRefMixin

```
makeBreakableRefMixin(parentObject) :: extend(parentObject, {
  erase(principal) :: {
    this().lostPrincipal(principal);
    joinedProviders: joinedbox.asVector();
    aSpareProviderIdx: joinedProviders.findFirst(lambda(spareProvider) -> {
      !(this().isBoundPrincipal(spareProvider))
    });
    if(!is_void(aSpareProviderIdx), {
      this().foundPrincipal(joinedProviders.get(aSpareProviderIdx))
    })
  }
});
```

First, the breakable reference signals the loss of the principal by invoking the `lostPrincipal` protocol method. The actual removal of the principal from the principal set is the responsibility of a cardinality mixin implementing that method. After having unbound the principal, the breakable reference tries to potentially rebind to spare candidate principals which are stored in the joinedbox. The `findFirst` method searches for a service provider in the joinedbox which is currently unbound and, when one is found, the breakable reference binds the spare candidate principal by invoking the `foundPrincipal` method to be implemented by a cardinality mixin. Note that, again, the cardinality mixin is given the guarantee that its principal argument is currently unbound.

3.3.5 Mixin Composition

Given the definition of the mixins depicted in figure 1, all different kinds of ambient references can be composed. For example, the constructor method shown below creates fragile ambient unireferences. The parameter `scope` denotes a table of either one argument, containing a required service type or of two arguments, containing a service type and the optional filter query.

```
makeFragileUniRef(scope) :: {
  makeFragileRefMixin(
    makeUniRefMixin(
      makeAmbientRefCore@scope))
}
```

The $1 + 3 + 3$ mixins (1 core, 3 cardinality mixins, 3 elasticity mixins) described above lead to a combination of $1 \times 3 \times 3 = 9$ different kinds of ambient references.

Their constructors have been made available via the `AmbientRefsMixin`. The mixin augments its behaviour object with access to the following constructor methods (arguments after the . are considered optional, this is done for readability purposes and is not regular AmbientTalk syntax):

```
AmbientRefsMixin
AmbientRefsMixin(obj) :: extend(obj, {
  ambientFragileUni(serviceType . filterQuery) :: { ... };
  ambientFragileMulti(cardinality, serviceType . filterQuery) :: { ... };
  ambientFragileOmni(serviceType . filterQuery) :: { ... };

  ambientElasticUni(elasticity, serviceType . filterQuery) :: { ... };
  ambientElasticMulti(elasticity, cardinality, serviceType . filterQuery) :: { ... };
  ambientElasticOmni(elasticity, serviceType . filterQuery) :: { ... };

  ambientSturdyUni(serviceType . filterQuery) :: { ... };
  ambientSturdyMulti(cardinality, serviceType . filterQuery) :: { ... };
  ambientSturdyOmni(serviceType . filterQuery) :: { ... };
  ...
})
```

Each such constructor function simply creates the right kind of ambient reference behaviour and wraps the resulting behaviour in a service actor. For example:

```
ambientFragileUni@args :: service(makeFragileUniRef(args));
```

4 Sustained Message Sends

Next to providing the constructor functions to create ambient references, the `AmbientRefsMixin` also introduces support for sustained message sends to ambient omnireferences. For example, sending a sustained zero-argument message `hello` to an omnireference `people` with a decay period of 10 seconds can be expressed using the ambient reference mixin as follows:

```
sustain(people#hello, [ ], 10*1000)
```

Implementation The implementation of this new language construct is as follows:

```
AmbientRefsMixin
forever :: -1;
sustain(msg, args, decayPeriod) :: {
  msg.setArgs(args);
  sustainedmsg : extend_with_args(msg, lambda(p) -> {
    decayPeriod :: p }, [decayPeriod]);
  send(sustainedmsg)
}
```

As can be witnessed from the above code excerpt, a sustained message send is a regular message send, where the message being sent is tagged with a slot `decayPeriod`, where an infinite decay period is encoded as a negative integer.

In order to support sustained message sends, ambient omnireferences check the `decayPeriod` slot of an incoming message to determine how long they should keep the message in the inbox. The remaining implementation of the `acceptMessage` method of the omnireference cardinality mixin is given below:

```
OmniRefMixin
acceptMessage(msg) :: {
  // broadcast message to all principals in the principal set
  ...
  if(msg.has_slot("decayPeriod"), {
    decayPeriod: msg.decayPeriod;
    if (decayPeriod > 0, {
      after(decayPeriod, lambda() -> {
        inbox.delete(msg)
      });
    })
    // else decayPeriod forever, leave it in inbox indefinitely
  }, {
  inbox.delete(msg) // regular messages deleted immediately
})
```

The omnireference first checks whether the message is tagged as a sustained message. If not, the message is deleted from the inbox immediately. If it is a sustained message, the ambient reference reads the decay period attached to the message and schedules a block of code for execution after the decay period has elapsed. This is done using the `after` language construct provided by the already mixed in `afterMixin` language mixin. The delayed code removes the message from the inbox. If the decay period is `forever`, no removal code is scheduled and the message is left in the inbox.

Because a new principal joining the principal set receives a copy of all messages in the inbox, sustained messages can be sent to principals that bind long after the sustained message has been received.

5 Snapshots

All ambient references encapsulate their principal set, making it impossible for clients to access, modify or enumerate the set of principals the reference is bound to at a certain moment in time. The main reason for doing so is that, for a number of ambient references (mostly the non-sturdy ones) the principal set is a volatile set which may change at any time as devices move about. Principals may join or leave the set at any time, hence also during an enumeration. In order to provide clean enumeration semantics, ambient references provide a "meta-level" `snapshot` operation.

The snapshot operation is "meta" because it is a message sent *to* an ambient reference rather than *via* an ambient reference to the principal(s) it denotes. By convention, meta-messages are prefixed by a μ[1]. The message μ`snapshot` may be sent to any kind of ambient reference and always returns a new sturdy multireference (referred to as *the snapshot*) initialized with the set of all bound principals of its argument reference and whose cardinality equals the size of this initial principal set. Because all of its principal slots are bound and sturdy, the snapshot will never bind with new services. The principal set of a fully bound sturdy multireference is by definition constant and can thus be safely enumerated by clients. It is never guaranteed that this enumeration accurately reflects the current availability of services. Elements of the snapshot can be disconnected but because the snapshot is sturdy, the bond with the service is maintained.

As an example, consider a PDA application that requires a printing service. Assume that a fragile omnireference named `printers` has been declared, denoting all available printing services on the network. In order to present the user with a list of all available services, one may enumerate a `snapshot` of the omnireference:

[1] In the current implementation, μ must be typed as `meta`.

```
availablePrinters : printers#μsnapshot();
tableOfPrinters : availablePrinters#μenumerate();
```

Implementation Snapshots are implemented by equipping every cardinality mixin with an additional `metasnapshot` method. We survey the different implementations for uni-, multi- and omnireferences below.

Unireference snapshots When making a snapshot of a unireference, the reference may either be bound or unbound. If the reference is unbound at the moment a snapshot is made, the result is an empty sturdy multireference with a cardinality of 0. If the reference is bound, the result is a unary sturdy multireference with a cardinality of 1. The implementation of a unireference's `metasnapshot` method is given below. The `makeSnapshot` function returns a new sturdy multireference with its principal table initialized to the given argument table. Its implementation is given below.

UniRefMixin

```
...
metasnapshot() :: { p: if(is_void(principal_), [], [principal_]); makeSnapshot(p) }
```

Multireference snapshots When a snapshot is made of a multireference, the result is a sturdy multireference whose principals correspond to all bound principal slots of the original multireference. Hence, if the original multireference has a cardinality of n and at most $k \leq n$ slots are bound at the moment the snapshot is made, the cardinality of the new multireference is k.

MultiRefMixin

```
...
metasnapshot() :: {
  boundPrincipals : vector.new();
  forEachSlot(lambda(slot,idx) -> {
    if(bound_slot(slot), {
      boundPrincipals.add(slot)
    })
  });
  makeSnapshot(boundPrincipals.asTable())
}
```

Omnireference snapshots When taking a snapshot of an ambient omnireference, the result is a sturdy multireference whose cardinality equals the size of the omnireference's principal set. However, when taking a snapshot of an omnireference, one may sometimes want to "postpone" the snapshot for a moment in order to allow the omnireference to "fill up" with useful service actors. In a sense, this requirement is related to the issue of sustained messages, where a client wants the omnireference to store the sustained message for a while in order to increase its chances of reaching a desired service.

It turns out that sustained message sends are againt the abstraction needed to solve the "delayed snapshot" problem. When sending the `metasnapshot` meta-message to an omnireference in a sustained manner, the snapshot will only be taken after the decay period of the message has elapsed. A good analogy exists between these delayed snapshots and the act of taking a real photograph with a camera. In a sense, the decay period corresponds to what is known as the *shutter speed* in photography. The shutter

speed is the time for which the shutter of the camera is held open during the taking of a photograph, to allow light to reach the film or imaging sensor. When more light is exposed to the sensor, the photograph will capture movement of objects, in a sense capturing a "longer" period of time. When taking a sustained snapshot of a sturdy omnireference, one achieves essentially the same effect, capturing more and more service actors as time progresses. Taking a snapshot with an infinite shutter speed is illegal.

OmniRefMixin

```
...
metasnapshot() :: {
  msg: thisMessage();
  if(msg.has_slot("decayPeriod"), {
    shutterSpeed: msg.decayPeriod;
    if (shutterSpeed > 0, {
      after(shutterSpeed, lambda() -> {
        makeSnapshot(principals_.asTable())
      })
    }, {
      error("snapshot made with an infinite decay period")
    })
  }, {
    makeSnapshot(principals_.asTable())
  })
}
```

Fixed Sturdy Multireferences In the previous paragraphs, the return value of each `metasnapshot` method is a snapshot created by invoking the `makeSnapshot` function. Its implementation is shown below:

```
makeSnapshot(principalTable) :: actor(makeFixedSturdyMultiRef(principalTable));
```

As can be witnessed from the above code, the resulting ambient sturdy multireference actor is not created by composing the multireference cardinality mixin with the sturdy reference elasticity mixin as one might have expected. Rather, a `FixedSturdyMultiRef` is returned. This ambient reference is actually an ad hoc, hardcoded implementation of sturdy multireferences. The reasons for introducing an ad hoc implementation are twofold:

- First, an ad hoc implementation of sturdy multireferences results in a huge efficiency gain. Because the principal set of a fully bound sturdy multireference will never change, the multireference has no need for a scope nor a service discovery mechanism, nor for the `foundPrincipal` and `lostPrincipal` methods. A sturdy multireference only has to forward each message it receives to all of its principals. Furthermore, as it knows that each principal slot is bound and will never become unbound, most of the checks in the cardinality and elasticity mixins become redundant.
- Second, sturdy multireferences can directly be initialized given a table of principals. The combination of mixins does not cater to such initialization.
- Third, a fixed sturdy multireference has a `metaenumerate` method which returns the encapsulated principal set. This operation is safe as the principal set is guaranteed to be constant.

The implementation of fixed sturdy multireferences is given below:

```
                         FixedSturdyMultiRefMixin
makeFixedSturdyMultiRef(principalTable) ::
 extend_with_args(root, lambda(principalTable) -> {
  principals_ : principalTable;
  ...
  in(msg) :: { copied from core object };
  acceptMessage(msg) :: {
    for(i:1, i<=size(principals_), i:=i+1, {
      to: principals_[i];
      cc : msg.copy();
      cc.setTarget(to);
      outbox.add(cc)
    });
    inbox.delete(msg)
  };
  μsnapshot() :: { makeSnapshot(principals_) };
  μenumerate() :: { principals_ }
}, [principalTable]);
```

The `AmbientReference` language mixin also provides the following constructor function to allow a client actor to explicitly construct a fixed sturdy multireference from a table of actor addresses. This is useful for constructing groups for the purposes of group (multicast) communication. The `ambientGroup` constructor is the only ambient reference constructor allowing for direct initialization of the principal set (i.e. it is the only ambient reference constructed from an external enumeration rather than from an intensional service type description).

```
ambientGroup@args :: { makeSnapshot(args) };
```

6 Multifutures

Ambient references support multifutures. Message sends to ambient references result in futures which represent the eventual return value. Messages sent to a multifuture acquired in this manner are forwarded by the multifuture to all of its resolved values. As more return values gradually become resolved, the multifuture sends all of the messages it has received thus far to a new resolved value such that the same messages are sent to all resolved values. Messages are only forwarded if the resolved value is an actor.

The value to which a multifuture resolves can be accessed using a `whenEach` language construct. The `whenEach` language construct takes a multifuture and a closure and schedules the closure for execution every time the future is resolved with a value. Note that it is never guaranteed how many times a `whenEach` observer will be invoked. This can be zero, one or more times and depends on the cardinality of the ambient reference and also on the availability of the principals. What *can* be guaranteed based on the cardinality is the maximum number of times `whenEach` is invoked: once for unireferences, n times for a multireference with cardinality n, an unbounded number of times for an omnireference.

For uni- and multireferences, it makes sense to execute a block of code when *all* of the principals have replied, because the number of replies is bounded. Hence, for multifutures returned by uni- and multireferences, one may subscribe a closure using `whenAll` to be invoked when all replies have arrived.

To support `whenAll`, the multireference cardinality mixin and the fixed sturdy multireferences are augmented with extra code to notify the multifuture of the maxi-

mum amount of resolved values it may expect. The extended code is shown below for multireferences:

```
MultiRefMixin
acceptMessage(msg) :: {
  if(!is_number(msg.getTarget()), {
    if(msg.has_slot("expectedNumberOfResults"), {
      msg.expectedNumberOfResults(cardinality)
    });
    ...
```

7 Ambient Reference Observers

The following code snippet shows how an actor can act as an "ambient sensor", monitoring the appearance and disappearance of any service of a certain type in the ambient:

```
AmbientSensor
AmbientSensor(aServiceType) :: service(extend_with_args(root, lambda(requiredType) -> {
  init() :: {
    // ambient* requiredType
    sensor: ambientFragileOmni(requiredType);
    when_found(sensor, lambda(ref) -> {
      display("Service online: ", ref, eoln);
    });
    when_lost(sensor, lambda(ref) -> {
      display("Service offline: ", ref, eoln);
    })
  }
}))
```

The language constructs `when_found` and `when_lost` both take an ambient reference (of any kind) and a code block as an argument. The code block is parameterized with the raw remote reference to the discovered or lost service actor. The code blocks are invoked whenever the argument ambient reference becomes bound or unbound. The semantics of binding and unbinding naturally depend on the kind of ambient reference the block of code is registered with. For example, adding a `when_lost` observer to a sturdy reference has no effect, as this kind of reference never lets go of its principals.

The two observer language constructs have as their return value a "subscription" object which can be used to cancel the subscription of the code blocks with their ambient references. By default, the code blocks are permanently registered with the ambient references. Once `cancel` is invoked upon a subscription object, it is guaranteed that the code block associated with the subscription will not be executed again. An example of cancelling a subscription is show below.

```
Subscription cancellation
  subscription : when_found(sensor, lambda(ref) -> {
    display("Service online: ", ref, eoln);
    subscription.cancel()
  });
```

Implementation The implementation of the ambient reference observers comprises three parts. First, the code of the core ambient reference mixin object is modified to incorporate a typical observer design pattern: the core object maintains a vector `__subscribers__` to hold both `when_found` and `when_lost` observers. Furthermore, the meta-interface of the core mixin is extended with two management methods

to add and delete observers: `metaSubscribeObserver` and `metaUnsubscribe-Observer`.

The core ambient reference object now also implements the methods `_notifyFound-Principal(p)` and `_notifyLostPrincipal(p)`. Invoking this method leads to the notification of all registered `when_found` or `when_lost` observers. The second part of the implementation of the ambient reference observers consists of calling these methods at the appropriate locations. This is done in each of the three cardinality mixins. Whenever a cardinality mixin binds with a new principal (in its `foundPrincipal` method) or loses a principal (in its `lostPrincipal` method), it invokes the appropriate notification method.

A third part of the implementation consists of extending the `AmbientRefsMixin` with the appropriate language constructs. This mixin keeps track of the registered closures via the map `__ambientRefObserverBlocks__` which maps integer IDs to closures. The integer IDs represent a unique identity for the ambient reference observers. This ID management is necessary because closures should not be parameter-passed directly to other actors – we pass their ID instead (cfr. the implementation of futures and `when`). These observers are implemented using the `ambientRefObserver` prototype. This object contains a `notify` method to be invoked by the ambient reference when the appropriate event takes place. It also provides a `cancel` method to unsubscribe itself with its reference.

The fixed sturdy multireferences explicitly disallow the registration of observers. Due to their nature, registered observers would never trigger anyway because fixed sturdy multireferences never bind anew or unbind.

References

[DVM+06] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer, 2006.

[VDM+06] Tom Van Cutsem, Jessie Dedecker, Stijn Mostinckx, Elisa Gonzalez, Theo D'Hondt, and Wolfgang De Meuter. Ambient references: Addressing objects in mobile networks. Technical Report VUB-PROG-TR-06-10, Vrije Universiteit Brussel, 2006.