Linguistic Symbiosis between Actors and Threads[⋆]

Tom Van Cutsem[⋆⋆], Stijn Mostinckx[⋆ ⋆ ⋆], and Wolfgang De Meuter
{tvcutsem|smostinc|wdmeuter}@vub.ac.be

Programming Technology Lab
Vrije Universiteit Brussel
Brussels – Belgium

Abstract. We describe a linguistic symbiosis between AmbientTalk, a flexible, domain-specific language for writing distributed programs and Java, a conventional object-oriented language. This symbiosis allows concerns related to distribution (service discovery, asynchronous communication, failure handling) to be handled in the domain-specific language, while still enabling the reuse of existing software components written in a conventional language. The symbiosis is novel in the sense that a mapping is defined between the concurrency models of both languages. AmbientTalk employs an inherently event-driven model based on actors, while conventional object-oriented languages employ a concurrency model based on threads. The contribution of this paper is a linguistic symbiosis which ensures that the invariants of the event-driven concurrency model are not violated by engaging in symbiosis with multithreaded programs.

Keywords: actors, threads, events, linguistic symbiosis, AmbientTalk

1 Introduction

The hardware advances in networking technology of the past decade have resulted in novel kinds of distributed systems, commonly referred to as *mobile ad hoc networks*. Such networks are no longer populated by large, immobile servers and desktop machines interconnected by reliable network cables, but rather are populated by small, mobile handheld computers or cellular phones interconnected by highly volatile wireless communication links. These hardware changes profoundly affect the design of software that has to run reliably in such an environment.

Our approach to tackle the novel distributed computing issues in mobile networks is based on dedicated programming language support. Language support can aid the programmer to build software that has been designed to run in such inherently volatile networks. Novel language constructs help the software developer both during the design phase (different programming paradigms foster different solutions to similar problems) as well as during actual development (dedicated language constructs can abstract away many low-level issues).

[⋆] Revised version accepted at the International Conference on Dynamic Languages (ESUG07).
[⋆⋆] Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)
[⋆ ⋆ ⋆] Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

We have developed a domain-specific language called *AmbientTalk* which provides built-in language constructs specifically geared towards distributed development for highly asynchronous, volatile, infrastructure-shy mobile ad hoc networks [1]. Naturally, the advantage of this approach is that programs related to AmbientTalk's problem domain can be more easily expressed. The downside of having a domain-specific language is that it is either not always appropriate to model regular application logic or simply that a lot of existing software components developed in other languages cannot be reused.

In this paper, we describe a *linguistic symbiosis* between the domain-specific, actor-based AmbientTalk language and the general-purpose object-oriented and multithreaded Java language. The symbiosis is profitable for both symbionts: AmbientTalk profits from the large amount of available Java software components, while Java profits from AmbientTalk's high-level support for distributed programming.

The main difficulty to be tackled by the AmbientTalk/Java symbiosis is the mapping of both languages' fundamentally different concurrency models. While AmbientTalk is entirely built on an event-driven actor-based architecture, Java employs a traditional multithreaded model. A linguistic symbiosis between two such models is not without danger: the event-driven model enforces certain concurrency constraints which could be violated by Java's multithreaded concurrency if Java objects access AmbientTalk objects without proper provisions. The contribution of this paper is a mapping between actors and threads which does not violate the properties of the event-driven actor model of AmbientTalk.

1.1 Motivation

Before describing the AmbientTalk/Java symbiosis and the problems that have to be tackled, we briefly highlight why AmbientTalk enforces an actor-based, event-driven model rather than a traditional multithreaded model. The first motivation for employing such a model has to do with the beneficial properties of actor-based distributed communication in the context of mobile ad hoc networks. In such networks, network partitions occur much more frequently because e.g. two mobile devices may move out of one another's wireless communication range. Such network partitions are often temporary: the partition is healed when the mobile devices move back into one another's communication range. The asynchronous nature of the actor model enables distributed communication to be decoupled in time by means of an actor's built-in message queues. For example, an object can send an asynchronous message to a disconnected object without being blocked because the message can be stored in a message queue while the recipient is disconnected. A more extensive discussion of the beneficial properties of actors in mobile networks can be found elsewhere [2].

The second motivation behind employing an event-driven concurrency model stems from the inherently event-driven nature of distributed systems. Devices may join or leave the network and messages can be received from remote devices at any point in time. In an event-driven concurrency model, event handlers implicitly denote an atomic block, restricting the non-determinism to the order in which events are processed. This is in contrast with pre-emptive multithreading where the programmer should manually denote such atomic blocks to ensure that the code is consistent with every possible interleaving of all threads in the system.

1.2 Problem Statement

We now describe the main concurrency problem that has to be tackled by the AmbientTalk/Java symbiosis. As will be explained in more detail in section 2.3, each AmbientTalk object is associated with *a single* actor and only this actor may directly invoke the object's methods. This ensures that race conditions on regular objects cannot occur. However, when an AmbientTalk object is passed as a parameter to a Java method invocation (via symbiosis), the object can become accessible to Java threads. When these threads in turn invoke a method on the AmbientTalk object (via symbiosis), the AmbientTalk object and any other accessible objects may be manipulated concurrently both by the object's actor and by other Java threads, violating the guarantee that race conditions on AmbientTalk objects cannot occur. As an example of how race conditions on objects could be caused, consider this following code which registers an AmbientTalk object as a Java `ActionListener` on an AWT `Button` instance.

```
actor: {
  // code executed by AmbientTalk actor
  def obj := object: { ... }
  button.addActionListener(object: {
    def actionPerformed(actionEvent) {
      // code executed by Java thread
    };
  });
}
```

The `actionPerformed` method is invoked by the AWT framework's thread when the button is clicked. Hence, objects visible to both the actor and the anonymous listener object, such as the `obj` object, could be manipulated by multiple concurrent threads, requiring the introduction of locks and other synchronisation constructs in AmbientTalk. In other words, AmbientTalk's actor model would have to be abandoned when engaging in symbiosis with Java. In section 4 a symbiotic protocol mapping is described that enables an event-driven actor language to safely engage in symbiosis with a multithreaded programming language, such that problems like the one illustrated above are avoided.

Although most prevalent object-oriented languages are multithreaded, that does not prevent programs written in those languages to foster an event-driven programming *style* (which is upheld by convention, not enforced by the language). Therefore the symbiosis described later pays specific attention to map the event-driven style of the thread-based language directly onto the appropriate event constructs of the event-driven programming language.

2 AmbientTalk

Before describing the AmbientTalk/Java symbiosis, we briefly introduce the AmbientTalk language, particularly emphasising its concurrency model. Although AmbientTalk is domain-specific in terms of its abstractions for distributed programming, it is a full-fledged object-oriented programming language in its own right. AmbientTalk inherits most of its standard language features from Self, Scheme and Smalltalk. From Scheme, it inherits the notion of true lexically scoped closures. From Self and Smalltalk, it inherits an expressive block closure syntax, the representation of closures as objects and the use of block closures for the definition of control structures. AmbientTalk's object model is derived from Self: classless, slot-based objects using delegation [3] as a reuse mechanism.

The concurrent and distributed features of the language have a different lineage. Rather than employing a multithreaded concurrency model, AmbientTalk's model is founded on the actor model of computation [4] and its many incarnations in languages such as Act1 [5], ABCL [6] and Actalk [7]. However, AmbientTalk's closest relative is the E programming language [8] (further described in section 7.1). E combines actors and objects into a unified model called *communicating event loops*, which is based on event loop concurrency, described in section 2.2.

2.1 Object-oriented Programming

The following code illustrates standard object-oriented programming in AmbientTalk.

```
def Account := object: {
  def balance := 0;
  def init(amount) { balance := amount };
  def deposit(amnt) { balance := balance + amnt };
  def withdraw(amnt) { balance := balance - amnt };
};
def LimitAccount := object: {
  super := Account;
  def limit := 0;
  def init(lowest, amount) {
    super :=  Account^new(amount);
    limit := lowest;
  };
  def withdraw(amnt) {
    (self.balance - amnt < limit).ifTrue: {
      raise: TransactionException.new(self, amnt);
    } ifFalse: {
      super^withdraw(amnt);
    }
  };
};
def account := LimitAccount.new(-500, 1000);
account.deposit(20);
account.balance; // returns 1020
```

Two prototypes are defined, one for `Account` objects and one for `LimitAccount` child objects, which set a limit to the amount of money that can be withdrawn from the account. Objects can be created ex-nihilo, by cloning or by instantiating objects. Instantiating an object is done by sending it the message `new`, which creates a shallow copy of that object and initialises the copy using its `init` method, which plays the role of "constructor". AmbientTalk's object instantiation is similar to class instantiation, except that the new object is a clone of an existing object, rather than an empty object allocated from a class.

Every object has a field slot named `super` denoting the "parent object" to which it delegates messages it cannot handle. The parent of an `Account` object is `nil`, the parent of a `LimitAccount` object is an `Account` object. Next to this implicit delegation, which occurs when an object receives a message it does not understand, AmbientTalk also allows objects to explicitly delegate a request to another object. The expression `obj^m()` delegates the message `m` to `obj`, leaving `self` bound to the sender. A traditional "super send" in AmbientTalk is then a message that is delegated to the object stored in an object's `super` slot.

Block closures are constructed by means of the syntax { `|args| body` }, where the arguments can be omitted if the block takes no arguments. Note that AmbientTalk supports both traditional canonical syntax as well as keyworded syntax for method definitions and invocations.

2.2 Event Loop Concurrency

AmbientTalk's concurrency model is based on communicating event loops [8]. In this model, an actor is represented as an event loop, rather than as a traditional "active object". An *event loop* is a thread of execution that perpetually processes *events* from its *event queue* by invoking a corresponding *event handler*. Communicating event loops enforce three essential concurrency control properties:

Property 1 (Serial Execution) *An event loop processes incoming events from its event queue one by one, i.e. in a strictly serial order.*

Property 1 ensures that the handling of a single event is atomic: race conditions on the event handler's state while handling the event cannot occur.

Property 2 (Non-blocking Communication) *An event loop* never *blocks waiting for another event loop to finish a computation. Rather, all communication between event loops occurs strictly by means of asynchronous event notifications.*

Property 2 ensures that event loops can never deadlock one another. However, in order to guarantee progress, event handlers should not execute e.g. infinite `while` loops. Long-running actions should be performed piecemeal by means of scheduling recursive events, such that an event loop always gets the chance to respond to other incoming events. An event loop can only be suspended when its event queue is empty.

Property 3 (Exclusive State Access) *An event loop has* exclusive *access to its mutable state. In other words, two or more event loops may* never *have direct access to shared mutable state.*

Property 3 ensures that event handlers never have to lock mutable state. Mutating another event loop's state has to be performed indirectly, by asking the event loop to mutate its own state via an event notification.

Event loop concurrency avoids deadlocks and race conditions *by design*. The non-determinism of the system is confined to the order in which events are processed. In standard pre-emptive thread-based systems, the non-determinism is much greater because threads may interleave upon each single instruction. In the following section, we describe how the abstract event loop model is incorporated into the AmbientTalk language.

2.3 AmbientTalk actors

In AmbientTalk, actors are represented as event loops: the event queue is represented by an actor's message queue, events are represented as asynchronous message sends, and event handlers are represented as the methods of regular objects. The actor's event loop thread perpetually takes a message from the message queue and invokes the corresponding method of the object denoted as the receiver of the message.

In AmbientTalk, each object is said to be *owned* by exactly one actor. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor communicate using standard, sequential message passing. It is possible for objects owned by an actor to refer to objects owned by other actors. Such references that span different actors are named *far references* (the terminology stems from E [8]) and only allow asynchronous access to the referenced object. Any messages sent via a far reference to an object are enqueued in the message queue of the object's owning actor and processed by that actor itself. AmbientTalk borrows from the E language the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). In the remainder of this paper, we assume that asynchronous sends do not return a value. Another semantics for return values is discussed in future work (see section 8).

Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the control flow of event loops which perpetually take messages from their message queue and synchronously execute the corresponding methods on the actor's owned objects. The control flow of an event loop never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. The message is enqueued and eventually processed by the receiver object's own actor.

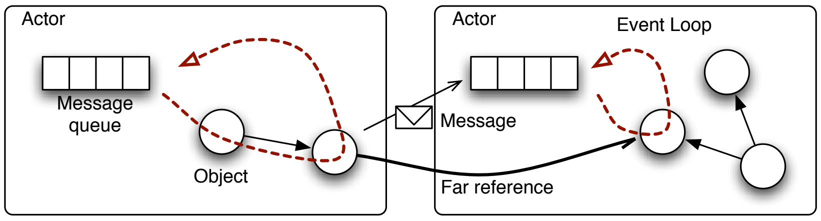

Fig. 1. AmbientTalk actors as communicating event loops

Because objects residing on different devices are necessarily owned by different actors, the only kinds of object references that can span across different devices are far references. This ensures by design that all distributed communication is asynchronous. AmbientTalk's far references are by default resilient to network disconnections: asynchronous messages may be buffered within the reference while the remote receiver object is disconnected. It is this design that makes AmbientTalk's distribution model suitable for mobile ad hoc networks because it allows one to abstract over temporary network partitions.

3 The AmbientTalk/Java Symbiosis

Our model for explaining the AmbientTalk/Java linguistic symbiosis is based on that of Inter-language Reflection [9]. In this model, a linguistic symbiosis consists of:

– a **data mapping** which ensures that data in one language looks like data in the other language, such that the symbiosis becomes as syntactically transparent as possible. For example, it is desirable that Java objects are equally represented as objects in AmbientTalk, such that messages can be sent to objects regardless of their native language.
– a **protocol mapping** between the meta-level representation of both languages' data. For example, both AmbientTalk and Java objects communicate by sending messages, but AmbientTalk is dynamically typed while Java is statically typed and exploits type overloading during method lookup. A proper symbiosis needs to map these message sending protocols onto one another.

AmbientTalk has been implemented in Java. Because of this, Java plays two roles: it is both a symbiont language and the implementation language of AmbientTalk (and hence of the linguistic symbiosis itself). Figure 2 illustrates the different objects that play a part in the AmbientTalk/Java symbiosis, according to the implementation model of Inter-language reflection [9]. AmbientTalk objects are physically implemented as Java objects. This is illustrated by means of the "represents" relationship. To enable symbiosis, additional objects are required which denote the *appearance* of objects from one language in the other language. At the implementation level, such appearances are implemented as *wrapper* objects, which wrap an object from a different language and which perform the protocol mapping which translates between the semantics of the symbiont languages. In what follows, we describe the standard data and protocol mappings for the AmbientTalk/Java symbiosis.

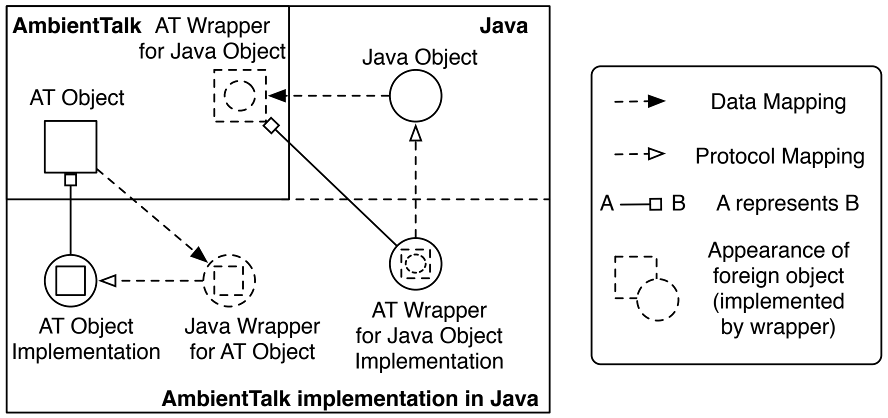
AmbientTalk
AT Wrapper for Java Object
Java
Java Object
AT Object
AT Object Implementation
Java Wrapper for AT Object
AT Wrapper for Java Object Implementation
AmbientTalk implementation in Java
Data Mapping
Protocol Mapping
A B
A represents B
Appearance of foreign object (implemented by wrapper)

Fig. 2. Symbiotic representation of AmbientTalk and Java Objects

3.1 Data Mapping

AmbientTalk's data mapping is similar to that of other dynamic languages implemented on top of the JVM such as Jython and JRuby (as discussed later in section 7.1). We introduce the linguistic constructs by means of a toy chat program, shown below. This small AmbientTalk program constructs a graphical user interface using the Java Swing framework. The GUI consists of a simple input field and an output text area. The code assumes that all connected chat participants are stored in a `participants` array. When the user enters text in the text field, AmbientTalk code broadcasts the text message to all connected participants. Note the registration of an anonymous AmbientTalk object as an action listener on the text field.

```
def swing := jlobby.javax.swing;
def JFrame := swing.JFrame;
def JTextField := swing.JTextField;
def JTextArea := swing.JTextArea;

// instantiate classes by sending them the "new" message
def frame := JFrame.new("Chat");
def textfield := JTextField.new(20);
def outputArea := JTextArea.new();

// static Java fields appear as fields on class wrapper
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// symbiotic method invocations
def pane := frame.getContentPane();
pane.setLayout(jlobby.java.awt.GridLayout.new(2,1));
pane.add(textfield);
pane.add(outputArea);
// the anonymous object is an AmbientTalk object that
// masquerades as a Java ActionListener object
textfield.addActionListener(object: {
  def actionPerformed(actionEvent) {
    // String returned by getText() is converted into AmbientTalk text
    def msg := textfield.getText();
    // participants is an array of all connected participants
    participants.each: { |chat| chat<-tell(msg) }
    // AmbientTalk text is turned into a String
    outputArea.append(msg);
  };
});
frame.setVisible(true);
```

Appearance of Java objects in AmbientTalk In order for AmbientTalk objects to talk to Java objects, they first need to get access to Java classes. From classes, objects can then be referenced via static fields or by instantiating the referenced classes. Java classes are organised hierarchically by means of packages. We have chosen to mimick this structural hierarchy by means of simple objects whose public slot names correspond to nested Java package or class names. The root of this hierarchy is named `jlobby`[1]. As can be seen from the code example, package objects can be created by selecting the slot with the appropriate name from `jlobby`.

[1] Ordinary AmbientTalk programs use an object called the `lobby` to load external objects, hence the name `jlobby` for loading Java classes.

Given a package object like the `swing` object in the chat example, it suffices to select the slot matching the Java class name to refer to a Java class as if it were an AmbientTalk object. Java classes appear as AmbientTalk objects whose fields and method slots correspond to public static fields and methods in the Java class. Hence, these fields and methods can be accessed or invoked using regular AmbientTalk syntax.

Java classes can be instantiated in AmbientTalk similar to how AmbientTalk objects are instantiated, i.e. by sending `new` to the wrapper for the class, which returns a wrapped instance of the Java class. Arguments to `new` are passed as arguments to the Java constructor. For example, in the chat application above, a new instance of a `JFrame` is created with the title of the frame passed as an AmbientTalk string. Java objects appear as AmbientTalk objects whose field and method slots correspond to public instance-level fields and methods in the Java object. These are accessed or invoked as if they were plain AmbientTalk slots.

The symbiotic architecture has a number of built-in conversions between several native datatypes. For example, AmbientTalk numbers are automatically converted into Java integers (e.g. `JTextField.`**`new`**`(20)`), AmbientTalk text is converted into Java `String`s, AmbientTalk arrays into Java arrays, AmbientTalk's `nil` value into Java's `null` value, etc. If the parameter-passed AmbientTalk object is actually a wrapper for a Java object, the unwrapped object is passed instead. These predefined conversions make the symbiosis highly transparent in most cases.

Appearance of AmbientTalk objects in Java There are two ways for Java code to gain access to AmbientTalk objects. The first is by embedding an AmbientTalk interpreter in existing Java code. The second is by means of a conversion rule when AmbientTalk code invokes a Java method which expects an argument typed as an *interface*. Any AmbientTalk object can be converted by the symbiosis into a Java object which *implements* that interface. Any messages sent by Java objects to this interface object are transformed into AmbientTalk message sends on the wrapped AmbientTalk object. In the chat example, the symbiotic call to `addActionListener` requires a parameter of type `ActionListener`, which is an interface type. Instead of passing a wrapped Java object implementing this interface, it is allowed to pass any AmbientTalk object; the object is not even required to implement all declared interface methods. The anonymous object passed in the above code properly implements the `actionPerformed` callback, and will be notified by Java code whenever the user has entered new text in the text field. A discussion on how the concomitant threading issues are avoided is postponed until section 4.

3.2 Protocol Mapping

For a linguistic symbiosis to work correctly, more is required than simply an appearance for the entities of the foreign language. A protocol mapping must be defined between the meta-level operations defined on the appearance of an object in the one language and the meta-level operations defined on the actual object in the other language [9]. In the case of AmbientTalk/Java, we consider three protocols: the slot access/method invocation protocol, which translates between AmbientTalk message sends and Java method invocations, the delegation/inheritance protocol, which translates between object delegation and class inheritance and the actor/thread protocol, which translates between event-driven and multithreaded concurrency control. We briefly describe the first two protocols below. The thread/actor protocol is described in detail in section 4.

The Slot Access/Method Invocation Protocol AmbientTalk's message sending protocol is entirely based on the concept that a selector (a symbol) uniquely denotes a slot in an object. In Java, on the other hand, static types can be used to *overload* a method name. At call time, the static types of the arguments are used to disambiguate the call. Another notable difference is that AmbientTalk supports explicit delegation, which implies that the object bound to `self` during method execution is not necessarily the object in which the method was found during method lookup. In Java, the value of `this` cannot be explicitly set to a separate delegating object. As a consequence, delegating a message to a Java object from within AmbientTalk does not allow the delegator to intercept self-sends performed by Java code.

Invoking Java methods in AmbientTalk The AmbientTalk/Java symbiosis treats message sends from AmbientTalk to Java as follows: if a message is sent to a class wrapper, only static fields or methods of the Java class are considered. If the message is sent to an instance wrapper, only non-static fields or methods of the Java class of the wrapped object are considered. If the AmbientTalk selector uniquely identifies a method (i.e. no overloading on the method name is performed in Java), the matching method is invoked. All AmbientTalk arguments are converted to Java objects by means of the data mapping described in the previous section. The Java return value is mapped back to an AmbientTalk value. If the Java method is overloaded based on arity (i.e. each overloaded method takes a different number of arguments), the number of arguments in the AmbientTalk invocation can be used to identify a unique Java method. If the Java method is overloaded based solely on argument types, the interpreter may derive that the actual arguments can only be converted from AmbientTalk to the appropriate Java types for exactly one of the matching overloaded signatures. In the remaining case in which the actual AmbientTalk arguments satisfy more than one overloaded method signature, the symbiotic invocation fails. It is then the AmbientTalk programmer's responsibility to provide explicit type information in the method invocation.

Invoking AmbientTalk methods in Java When an AmbientTalk object is passed as an argument to a Java method expecting an object of an interface type, the AmbientTalk object will appear to Java objects as a regular Java object implementing that interface. Hence, messages sent to this wrapped AmbientTalk object appear as regular Java method invocations on an interface type. For example, the action listener in the chat example can be notified as if it were a normal Java object by performing `listener.actionPerformed(event)`.

If Java invokes a method declared in an interface with an overloaded method signature, all overloaded invocations are transformed into the same method invocation on the AmbientTalk object. In other words, the AmbientTalk object does not take the types into consideration. However, if the Java method is overloaded based on arity, the AmbientTalk programmer can take this into account in the parameter list of the corresponding

AmbientTalk method, by means of a variable-argument list or optional parameters. Otherwise, the Java invocation may fail because of an arity mismatch.

The Delegation/Inheritance Protocol In AmbientTalk, when objects need shared access to state or behaviour, they can do so by designating an object to hold that state or behaviour as their common parent. This use of delegation was first advocated by Lieberman [3] and is a key programming pattern in the prototype-based language Self. In Self, these shared parent objects are called *traits* and they play the role of shared repositories of behaviour normally played by classes in class-based languages [10].

The AmbientTalk/Java symbiosis represents the instance-class relationship of Java objects by means of the delegation link of the AmbientTalk wrappers: the `super` slot of an AmbientTalk wrapper for a Java object always points to the wrapper of that Java object's class. Because of this design, the concept of a class in Java symbiotically appears as the concept of a trait in AmbientTalk.

The delegation relationship between AmbientTalk objects is not mapped to Java inheritance concepts when passing an AmbientTalk object to Java code. To Java code, the AmbientTalk object appears as an instance of a class implementing some interface. Java interfaces may extend other interfaces. It may be that the AmbientTalk object implements those extended interfaces by delegating to other AmbientTalk objects, but this is an implementation detail, just like it is an implementation detail how a Java class implements the methods of its declared interfaces.

4 The Actor/Thread Protocol

In this section, we describe a symbiotic protocol mapping for representing AmbientTalk actors as Java threads, and more interestingly, for streamlining multithreaded concurrency in Java as asynchronous message sending in AmbientTalk.

4.1 Viewing Actors as Threads

As described previously, an AmbientTalk actor consists of a message queue, an event loop and a number of objects owned by the actor. The event loop is essentially a thread which perpetually reads the next message from the message queue and invokes the method on an owned object designated as the receiver of the message. The invoked method is executed by the actor's event loop thread.

Symbiotic invocations on Java objects If an AmbientTalk object performs a symbiotic invocation on a Java object, the symbiotically invoked Java code is still executed by the actor's event loop thread. From a symbiosis point of view, an actor in AmbientTalk appears as a thread in Java by representing the actor as its own event loop thread. When an actor appears as a thread at the Java level, it has to abide by the rules of shared-state multithreaded concurrency: the event loop thread may take locks on Java objects and use Java's `wait` and `notify` synchronisation primitives. When the symbiotic invocation

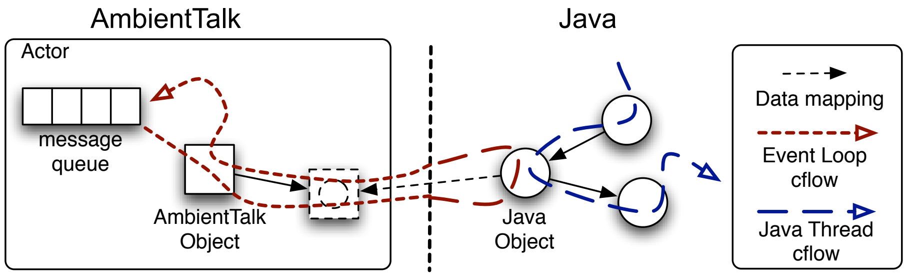

Fig. 3. Representing AmbientTalk actors as Java threads

finally returns, the event loop thread transparently starts executing AmbientTalk code again and is trivially converted into the event loop of an actor again.

Figure 3 illustrates the control flow of passing from the AmbientTalk level to the Java level. On the left-hand side of the figure, an AmbientTalk actor is executing a symbiotic invocation on a wrapped Java object. Notice how the event loop thread flows from the AmbientTalk into the Java level to execute the invoked Java method. Because other Java threads may be concurrently operating on the same object, synchronisation between the threads may be required.

Immediate symbiotic invocations When an AmbientTalk actor appears as a Java thread, it is possible that the thread performs a symbiotic invocation on a wrapped AmbientTalk object. For example, consider an AmbientTalk visitor object that is passed as argument in the `accept` method of a Java object which calls back on the visitor to visit the appropriate type. To enable a Java thread representing an actor to perform invocations on AmbientTalk objects, the symbiosis must define an appropriate equivalent in Java for the "ownership" relationship between objects and actors in AmbientTalk.

When an event loop thread enters the Java level, the symbiosis considers any Java object on which the thread operates as being owned by the actor represented by the thread. Hence, Java objects which are reachable from objects owned by an AmbientTalk actor are transitively considered owned by that actor. When multiple event loop threads access the same Java object, this object becomes owned by multiple actors, which may lead to ill-defined behaviour (see below). When a Java object that is transitively owned by an actor invokes a method on a wrapped AmbientTalk object, the protocol is as follows:

- If the owner of the wrapped AmbientTalk object equals the owner of the Java object, the invocation is performed *immediately*, by the Java thread representing the event loop itself. There is no need to synchronise access with the AmbientTalk actor's event loop because the Java thread *is* that event loop. If access would be synchronised, the event loop would wait for itself, resulting in immediate deadlock.
- If the owner of the wrapped AmbientTalk object does not equal the owner of the Java object, this implies that an AmbientTalk actor has gained direct access to an AmbientTalk object owned by another actor. Hence, the actor has circumvented

the exclusive state access property by sharing an object with another actor at the Java level. In this case, the interpreter aborts the symbiotic invocation, rather than allowing race conditions to occur between the actors. The alternative of synchronising access to the AmbientTalk object is not viable, as this would violate the non-blocking communication property: one event loop (the executing Java thread) would be blocked waiting for another event loop (the event loop of the actor owning the wrapped object).

4.2 Viewing Threads as Actors

We now describe the protocol mapping that allows the threading model of Java to coexist with the actor model based on communicating event loops of AmbientTalk. The symbiosis distinguishes between synchronous and asynchronous symbiotic invocations on AmbientTalk objects by Java code.

Synchronous Symbiotic Invocations When a Java thread (which does not represent an AmbientTalk actor) performs a method invocation on a wrapped AmbientTalk object, the method invocation *cannot* be executed immediately by the Java thread. Doing this would violate the serial access property of event loop concurrency, which ensures that no race conditions can occur on objects owned by an actor.

The solution employed by the AmbientTalk/Java symbiosis is to regard a Java wrapper for an AmbientTalk object as a *far reference* to the actual AmbientTalk object, and to interpret every Java method invocation on that wrapper as an *asynchronous message send*. The Java thread schedules the symbiotic method invocation for asynchronous execution in the message queue of the actor owning the wrapped AmbientTalk object, rather than synchronously invoking the method on the actual AmbientTalk object itself. Hence, it is the actor's own event loop that will process the symbiotic invocation, ensuring that the serial access property of the event-loop model remains intact.

By turning synchronous Java invocations on wrapped AmbientTalk objects into asynchronous AmbientTalk message sends, the properties of AmbientTalk's actor model remain intact. However, Java's threading model based on synchronous method invocation cannot always deal with this asynchrony: a method invocation normally returns a value or it may throw an exception. To reconcile asynchronous message sends with synchronous method invocations, we employ *futures*, which are a well-known abstraction that represent a handle to the return value of an asynchronous request (see section 7.3).

The asynchronous scheduling of the symbiotic invocation immediately returns a future object to the Java thread that schedules the request. The Java thread can then (explicitly) synchronise on the future, suspending the thread until the future either gets *resolved* with a return value, or until it is *ruined* by an exception (raised in the asynchronously invoked AmbientTalk code). When the event loop of an actor dequeues a symbiotic invocation request, it invokes the AmbientTalk method and uses the return value (respectively a caught exception) to resolve (respectively ruin) the future attached to the symbiotic invocation. This wakes up the waiting Java thread, which can then return from the symbiotic method invocation.

Figure 4 illustrates the appearance of Java wrappers for AmbientTalk objects as far references at the AmbientTalk level. On the left-hand side of the figure, a Java thread

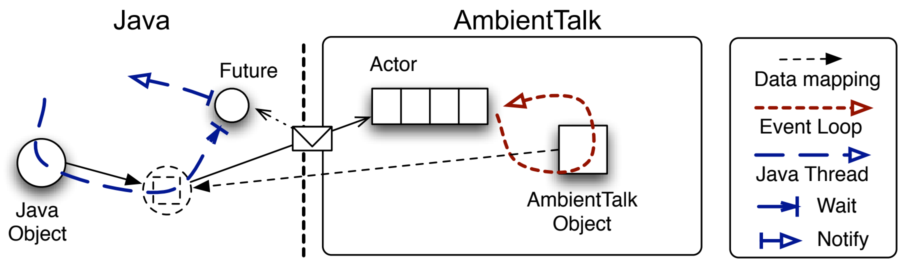

Fig. 4. Java method invocations appear as AmbientTalk message sends

is performing a symbiotic invocation on a wrapped AmbientTalk object. Rather than making the Java thread enter the AmbientTalk actor, a symbiotic invocation is scheduled in the message queue of the wrapped object's actor. The Java thread synchronises on the associated future object. While the Java thread is blocked, the actor event loop processes its incoming messages. When the symbiotic invocation has been completed, the future object is notified and the symbiotic invocation returns, enabling the Java thread to proceed its execution. At the Java level, the call appeared to be synchronous, at the AmbientTalk level, it appeared to be an asynchronous message send.

Asynchronous Symbiotic Invocations Java's native concurrency model is based on shared-state concurrency, i.e. multiple threads communicating by modifying shared objects. For many applications, this concurrency model is inappropriate. Many interactive applications (e.g. games, user interface frameworks) or discrete-event simulations require an event-driven approach. In Java, event-driven programming cannot be enforced. However, an event-driven style can be adopted by using an event loop framework. The Java AWT and Swing toolkits are the quintessence of such an approach.

In event-driven Java frameworks, asynchronous message sends are necessarily represented implicitly by synchronously invoking the methods of so-called *listener* objects. The documentation of most event-driven Java frameworks specifies that such methods must return as soon as possible, and should preferably only schedule tasks for later execution. In effect, this means that asynchronous message passing in Java is implemented as a *second-class* language construct by means of listener invocations. However, our symbiosis can detect such second-class asynchronous message sends and map them onto *actual* asynchronous sends in AmbientTalk, without having to synchronise the Java thread. This is highly desirable because it ensures the responsiveness of the event-driven Java framework.

As an example, consider an AmbientTalk object that is registered as an `Action-Listener` on an AWT `Button`. When the AWT event loop invokes the `action-Performed` method on the AmbientTalk object, this implicitly indicates an event notification, which is conceptually asynchronous. Hence, rather than making the AWT event loop suspend until the AmbientTalk actor has actually processed the `actionPer-formed` method, as is the case for a synchronous symbiotic invocation, the method invocation is turned into an actual asynchronous message send, and the event loop thread can return immediately.

The AmbientTalk/Java symbiosis treats the invocation of any method belonging to (an extension of) the `java.util.EventListener` interface as an asynchronous message send, provided the method has a `void` return type and does not declare any thrown exception. It is a convention of Java frameworks that objects representing event listeners be tagged with (a subtype of) this interface. When a method on a wrapped AmbientTalk object representing an `EventListener` is invoked, a symbiotic call is scheduled as described previously, but no future is created on which to synchronise the Java thread. Rather, the Java thread returns immediately. Hence, the Java method invocation `buttonListener.actionPerformed(event)` is effectively interpreted by AmbientTalk as `buttonListener<-actionPerformed(event)`.

The symbiosis described above only covers asynchronous message sends without return values. A discussion on the possibility of allowing symbiotic future-type message sends is postponed until section 8.

4.3 Summary

In this section, we have described how actors and threads interact. When AmbientTalk objects invoke methods on Java objects, the invocation is always synchronous. When Java objects invoke methods on AmbientTalk objects, we distinguish three different kinds of symbiotic method invocations:

- Immediate symbiotic invocations occur when a Java thread that represents an AmbientTalk actor calls back on its own objects.
- Synchronous symbiotic invocations occur when a regular Java thread invokes a method on an AmbientTalk object.
- Asynchronous symbiotic invocations occur when a regular Java thread invokes a method that represents an event notification on an AmbientTalk object (i.e. the AmbientTalk object acts as a listener).

5 Applications

We now give concrete examples of each of the three symbiotic invocations on AmbientTalk objects described in the previous section, thereby illustrating how the thread/actor protocol mapping behaves in practice.

5.1 Immediate Invocation

A traditional advantage of introducing symbiosis between a small language like AmbientTalk and an industry-strength language like Java is that the vast amount of libraries available in the latter language can be reused. However, the use of a Java library is hardly ever restricted to simple one-way calls from AmbientTalk to Java. In many Java frameworks, for example, the framework objects call back on the parameter-passed AmbientTalk objects. We illustrate this interchange of messages by means of the Java

Collection Framework. In the example, an AmbientTalk program periodically receives address card objects from PDAs in its environment. It will store these objects in a set to filter out duplicates. Moreover, the address cards should be ordered by name so that the user can be presented with an alphabetical overview of all nearby persons. This can be achieved with the following AmbientTalk code excerpt:

```
def vCardPrototype := object: {
  // implements the java.lang.Comparable interface
  def compareTo(vCard) {
    self.fullName.compareTo(vCard.fullName);
  };
  // define fields for fullName, address, etc.
};
def contacts := jlobby.java.util.TreeSet.new();
whenever: VCard discovered: { |vCard|
  contacts.add(vCard);
};
```

Whenever a vCard object is discovered in the network, the `add` method is invoked on the wrapped `TreeSet`. In order to correctly insert the element, the set repeatedly invokes the `compareTo` method on the Java wrapper for the inserted `vCard` object. As the thread invoking the method is the event loop thread owning the wrapped AmbientTalk object, the method can be executed immediately.

The immediate invocation presented in this section is the most common form of symbiotic invocations, often being the result of passing AmbientTalk objects as parameters to Java libraries. The Visitor design pattern [11] can be seen as the epitome of such an interaction. When traversing a Java data structure with an AmbientTalk visitor, a double dispatch across the language boundary occurs between `accept` and `visitType` messages. Such examples illustrate why it is necessary for the thread/actor protocol mapping to distinguish immediate invocations from synchronous invocations. If the callback operation from Java to AmbientTalk is not executed immediately, but rather by means of a synchronous invocation, the AmbientTalk actor would wait for itself, resulting in immediate deadlock.

5.2 Synchronous Invocation

As noted previously, linguistic symbiosis is often useful for a small language like AmbientTalk to reuse the large amount of software components available in a language like Java. However, it is equally viable for the Java programmer to embed AmbientTalk components into an existing Java framework. This allows the Java programmer to profit from e.g. AmbientTalk's language support for distributed programming.

In this section, we illustrate how AmbientTalk unit tests can be incorporated into the JUnit unit testing framework, allowing a Java developer to integrate all unit tests in a consistent testing framework. In AmbientTalk, a unit test is an object whose methods are prefixed with `test`. All unit test objects delegate to the prototypical unit test object, which contains reflective code to invoke all test cases. Note that this object also implicitly implements the `junit.framework.Test` interface.

```
def UnitTestPrototype := object: {
  def testMethods; // array of first-class method objects
  def init() {
    testMethods := retrieveTestMethods(self);
  };
  def countTestCases() { testMethods.length };
  def run(reporter) {
    reporter.startTest(self);
    testMethods.each: { |method| /* perform the test */ };
    reporter.endTest(self);
  };
}
```

Now consider a `TestSuite` that is composed of both unit tests written in Java and unit tests written in AmbientTalk. All unit tests uniformly implement the `Test` interface. In order to incorporate an AmbientTalk unit test into the test suite, it suffices to wrap the AmbientTalk object representing the unit test explicitly in the `Test` interface. The following code excerpt assumes that `exampleATTest` is a reference to an AmbientTalk unit test object.

```
public static void main(String[] args) {
  TestSuite suite = new TestSuite();
  ATObject exampleATTest = /* load AmbientTalk test */;
  Test exampleJavaTest = /* load Java test */;
  suite.addTest((Test) wrap(exampleATTest, Test.class));
  suite.addTest(exampleJavaTest);
  junit.textui.TestRunner.run(suite);
}
// see section 6 for details on wrapping AmbientTalk objects
static Object wrap(ATObject obj, Class interface) {
  return Proxy.newProxyInstance(interface.getClassLoader(),
                                new Class[] { interface },
                                new JavaWrapperForATObject(obj));
}
```

A `TestRunner` executes the test suite by sequentially invoking each unit test's `run` method. This execution is performed by a Java application thread. Because an embedded AmbientTalk unit test should be run inside its owning actor, an invocation of the `run` method on the wrapped AmbientTalk unit test schedules an asynchronous request in the actor owning `exampleATTest`. However, the JUnit framework expects `run()` to be a synchronous invocation, implying that the Java thread should obviously wait for the test to be completed before executing the next test or terminating. Therefore, the Java thread suspends transparently until the AmbientTalk actor has processed the `run` method.

Synchronous invocation is enforced when a Java thread (which does not represent an actor) performs invocations on a wrapped AmbientTalk object. This is typically the case when using AmbientTalk from within a Java application, or when passing AmbientTalk objects to libraries which internally start their own threads.

5.3 Asynchronous Invocation

As described in section 4.2, many Java applications are themselves event-driven. We have already shown how AmbientTalk objects can engage in a proper symbiosis with event-driven frameworks without any problem. AmbientTalk objects may implement event listener interfaces whose methods are invoked purely asynchronously. In this section, we illustrate another use of asynchronous symbiotic invocations. The goal is for AmbientTalk code to be able to reuse Java's `java.util.Timer` abstraction. The `Timer` class is typically used to schedule tasks for execution at a later point in time as follows:

```
Timer timer = new Timer();
TimerTask task = new TimerTask() {
  public void run() {
    System.out.println("executing task");
  }
}
// schedule task to be executed in 5 sec
timer.schedule(task, 5000);
```

The `run` method of the `TimerTask` instance is invoked by the `timer` object after 5000 milliseconds have elapsed. In Java, this callback is executed by the thread of `timer`, which is not necessarily the thread that scheduled the task. Hence, explicit synchronisation between both threads is often necessary to prevent race conditions.

In AmbientTalk, we would like to be able to schedule code for execution at a later point in time, without causing race conditions in the actor. Unfortunately, `TimerTask` is not an interface but an abstract Java class, so it cannot be instantiated. Neither can AmbientTalk code create a concrete subclass of `TimerTask`. To make the symbiosis work, a small auxiliary class needs to be written in Java:

```
public class ATTimerTask extends TimerTask {
  public interface AsyncRunnable extends EventListener {
    public void run();
  }
  private AsyncRunnable code;
  public ATTimerTask(AsyncRunnable r) {
    code = r;
  }
  public void run() {
    code.run();
  }
}
```

The above class defines a simple wrapper around an object that understands the message `run`. By means of this auxiliary class, it is easy to wrap AmbientTalk objects in Java `TimerTask` instances:

```
def timer := jlobby.java.util.Timer.new();
def task := jlobby.at.support.ATTimerTask.new(
  object: {
    def run() { system.println("executing task") }
  });
timer.schedule(task, 5000);
```

The above code assumes that the class `at.support.ATTimerTask` is available on the JVM's class path. The thread/actor protocol mapping ensures that the `run` method of the anonymous AmbientTalk object is executed by its owning actor, *not* by the thread of the timer. Moreover, because the `AsyncRunnable` interface has been marked as an `EventListener` interface, the AmbientTalk/Java symbiosis treats the `code.run()` call in the auxiliary Java class as an asynchronous send. This ensures that the Java timer thread is *not* blocked waiting for the AmbientTalk actor to process the invocation, such that it can timely execute other scheduled tasks.

The AmbientTalk/Java symbiosis allows AmbientTalk code to use Java's `Timer` framework in exactly the same way as the framework would be used in Java, except that the multithreaded concurrency of Java is automatically adapted to the event-driven concurrency of AmbientTalk. The symbiosis layer ensures that this mapping is done transparently, without additional programming effort in AmbientTalk itself.

6 Implementation

This section describes the detailed implementation of the thread/actor protocol mapping introduced in section 4.2. The thread/actor mapping occurs in the Java wrapper of an AmbientTalk object. This Java wrapper is implemented by means of Java's standard support for dynamic proxies. When an AmbientTalk object is passed as an argument to a Java method requiring a parameter of an interface type, a dynamic proxy implementing that interface is generated by the symbiosis. The proxy requires an object implementing the `InvocationHandler` interface whose `invoke` method is used to intercept messages sent to the proxy. The `JavaWrapperForATObject` class implements Java wrappers for AmbientTalk objects. The essential part of the code is shown below.

```java
class JavaWrapperForATObject implements InvocationHandler {
  final ATObject principal; // the wrapped AmbientTalk object
  final EventLoop owningEventLoop; // the thread of the owner actor
  JavaWrapperForATObject(ATObject atObj) {
    principal = atObj;
    owningEventLoop = EventLoop.fromThread(Thread.currentThread());
  }
  Object invoke(Object rcv, final Method method, Object[] args) throws Throwable {
    final ATObject[] atArgs = new ATObject[args.length];
    for (int i = 0; i < atArgs.length; i++) {
      atArgs[i] = Symbiosis.javaToAmbientTalk(args[i]);
    }
    Event symbioticInvocation = new Event() {
      public Object process() throws Exception {
        ATObject result = Symbiosis.downInvocation(principal, method, atArgs);
        return Symbiosis.ambientTalkToJava(result, method.getReturnType());
      }
    }
    if (owningEventLoop.equals(Thread.currentThread())) { // immediate invocation
      return symbioticInvocation.process(); // immediately perform invocation
    } else { // detecting exclusive access violations
      if (EventLoop.isEventLoop(Thread.currentThread())) {
        throw new RuntimeException("Violated exclusive access property");
      }
      if (Symbiosis.isEvent(method)) { // asynchronous invocation
        owningEventLoop.schedule(symbioticInvocation);
        return null; // void return type
      } else { // synchronous invocation
        Future f = new Future(); // to make the Java thread wait for the result
        owningEventLoop.schedule(symbioticInvocation, f);
        return f.get(); // blocks until invocation has been executed by actor
32       }
33     }
34   }
35 }
```

A `JavaWrapperForATObject` wraps an AmbientTalk object (the `principal`) and the event loop thread of the actor that owns that object (the `owningEventLoop`). The implementation assumes that the thread creating the wrapper is the event loop thread of the owning actor (line 6).

When a method is invoked on the dynamic proxy, the Java arguments to the call are mapped onto AmbientTalk values (lines 9-12). The data mapping is performed by means of the `javaToAmbientTalk` method. Subsequently, an event object is created which can be scheduled in an actor's event queue (lines 13-18). When the event is processed, it transforms the Java method invocation into an AmbientTalk invocation, according to the protocol mapping outlined in section 3.2. The AmbientTalk return value is mapped to a Java value of the appropriate return type (line 16). The rest of the code implements the thread/actor protocol mapping. It is subdivided into four parts:

Immediate invocation First, it is checked whether the symbiotic invocation is performed by the owning actor's own event loop thread (line 19). If this is the case, the symbiotic invocation is performed immediately using the current thread. This ensures that e.g. the callback messages described in section 5.1 operate correctly.

Detecting exclusive access violations By providing AmbientTalk actors with the power to access Java objects, we have introduced the problem that two or more actors may share their objects via the Java level, bypassing the exclusive access property. However, by treating Java objects as being "owned" by the thread that operates on them, we can detect when the event loop thread of one actor tries to access an object owned by another actor. In the code, it is checked whether the accessing thread is the event loop of another actor (line 22). If this is the case, an AmbientTalk actor has gained access to an object owned by another actor, and the symbiotic invocation is aborted.

Asynchronous invocation If the thread performing the symbiotic invocation is a regular Java thread, but the invoked method represents an event notification (i.e. it belongs to an `EventListener` interface, has a `void` return type and does not declare any thrown exceptions), the invocation is scheduled asynchronously and the thread returns immediately (lines 25-27). The `schedule` method places the event in the event queue of the owning actor. This queue is served by the actor's own event loop thread which will eventually invoke the `process` method of the scheduled `Event` object.

Synchronous invocation In the final case, the symbiotic invocation is performed by a regular Java thread on a non-event method (lines 29-31). In this case, the Java thread must wait until the owning actor has processed the invocation. This synchronisation is performed by means of a small auxiliary `Future` class. A future is passed to the actor's `schedule` method which will be resolved by the actor with the result of processing the scheduled `Event`. The Java thread subsequently suspends and waits for this result by invoking the future's `get` method.

Performance Measurements To determine the performance of symbiotic method invocations from Java to AmbientTalk, we have compared the overhead of synchronous symbiotic invocations (which require the Java thread to schedule an event and suspend on a future) with respect to immediate invocations (which allow the calling thread to immediately execute the symbiotic invocation). The results are shown below[2]. Three kinds of methods are invoked: a method with an empty body and two calculating a linear function requiring 25 resp. 50 iterations. The methods are invoked by Java code ran by the AmbientTalk actor itself (immediate invocation) and by an auxiliary Java thread (synchronous invocation).

	Runtime (in milliseconds) of		
	Empty Method	For-loop 25	For-loop 50
Immediate	3.24	57.35	112.51
Synchronous	4.36	58.39	113.19
Difference	1.12	1.04	0.68
Overhead	25.69%	1.78%	0.60%

The results show the expected result that synchronous invocations are slower than immediate invocations. Especially for very small methods this overhead is significant. Naturally, the overhead quickly decreases when methods take longer to execute. On average, the overhead introduced by synchronous invocations is .94 milliseconds. One important point is that all synchronous symbiotic invocations were performed on an actor whose message queue was *empty*. As the load of an actor increases, it takes more time before the synchronous invocation is served from the queue which of course drastically influences the runtime of the synchronous invocation.

7 Related Work

7.1 Linguistic Symbiosis

The reference model that we used for explaining linguistic symbiosis is that of Interlanguage Reflection [9] which is itself based on an open design of object-oriented languages [12]. The purpose of these models is to clearly identify which objects define the boundaries between two languages (data mapping) and to identify consistent rules for crossing the boundaries (protocol mapping).

Performing a linguistic symbiosis between two languages in order to combine different programming paradigms is not novel. For example, SOUL/Smalltalk is a linguistic symbiosis between a logic programming language and an object-oriented language, where the goal is to use the logic language to *reason about* the object-oriented language [9, 13].

There exist a vast number of dynamic languages that have been implemented on top of the Java Virtual Machine and which provide a symbiotic layer to interface with Java objects. Examples include JRuby, Jython, JScheme, LuaJava and JPiccola. The data

[2] Average runtime of 300 invocations on an AmbientTalk (version 2.5) object by a Java thread as measured on an Apple Powerbook G4 1Ghz using the HotSpot JVM v1.3.1 on Mac OS X.

and protocol mappings of these languages is often similar in spirit, differing only in the details of e.g. how method overloading is handled. Some implementations are more advanced than others. For example, Jython allows Python classes to subclass Java classes. To the best of our knowledge, none of these linguistic symbioses define a concurrency protocol mapping, often because the two symbionts employ a similar thread-based concurrency model.

Piccola [14] is a *composition language* designed to glue together components from its underlying *host* language. Hence, Piccola requires a linguistic symbiosis with its host language in order to access and manipulate the language's components. For example, JPiccola – which uses Java as the host language – defines a so-called composition style that allows Java AWT components to be easily composed in Piccola. Piccola's concurrency control is based on the π-calculus, where agent processes communicate via channels. To the best of our knowledge, Piccola does not define a mapping between its host language's concurrency model and the agent and channel abstractions of the π-calculus, although the Piccola programmer may define such a mapping himself by defining his own composition style.

AmbientTalk's event loop concurrency model is based on that of the E language [8]. E is a language for capability-secure distributed computing in open networks. E also provides symbiotic access to Java, the language in which it is implemented. However, to the best of our knowledge, E does not provide a concurrency protocol mapping as described in this paper. The only documented feature in this regard is E's ability to allow one of its event loops to *morph into* an AWT or Swing event loop thread. In this way, all threads in the system correspond to E event loops, which effectively rules out a coexistence of the thread-based concurrency model of Java with the communicating event loops of E. This solution can be regarded as a specific instance of AmbientTalk/Java's asynchronous symbiotic invocations.

7.2 Unifying threads and events

The Scala language offers a concurrency library based on actors that unify threads and events [15, 16]. Scala actors are equipped with two kinds of message reception mechanisms. Thread-based actors have a `receive` operation that suspends the actor's thread until a suitable message specified in the `receive` block arrives in the actor's mailbox. When such a message finally arrives, the thread is resumed and the `receive` operation returns normally. Event-based actors have a `react` operation which, when no suitable message is available in the mailbox, does not suspend the current thread, but rather stores the actor's `react` code as a closure and signals to the executing thread that the actor is "suspended" by throwing a special exception. This exception allows the executing thread of the event-based actor to continue executing other actors. Scala's approach differs from ours in the sense that they provide one unified concurrency model, while we provide a language symbiosis that bridges the distinct concurrency models of two separate languages. Also, the actor model of Scala is more liberal than that of AmbientTalk, in the sense that the three properties of event loop concurrency are not enforced by Scala's model. It is the programmer's own responsibility to guarantee that Scala actors do not perform concurrent updates on shared state but communicate only by message passing.

Li and Zdancewic describe a concurrency model that unifies threads and events, implemented in Haskell [17]. In their approach, the system offers two views on multi-threaded programs: a thread view, which allows regular application code to program in a thread-based way, and an event view, which allows programmers to manipulate threads as passive entities by means of an event-driven thread scheduler. Technically, threads are cut into a series of event handlers at points where the code performs blocking I/O operations. The difference with our approach lies again in the fact that a unified model is presented, not a symbiosis between two languages with a separate model. Also, Li and Zdancewic stress the use of their event-driven model to write custom thread scheduling code, to achieve better performance. In our approach, the event-driven model is meant to be used for actual application-level code in order to avoid race conditions and deadlocks by design in highly volatile distributed systems.

7.3 Futures

Our synchronisation technique between synchronous Java invocations and asynchronous AmbientTalk message sends is essentially an application of futures [18]. In concurrent object-oriented programming languages, futures have been a recurring language abstraction to reconcile asynchronous message passing with return values (e.g. future-type message passing in ABCL [6], promises in Argus [19] and E [8], wait-by-necessity in Eiffel// [20]). A future is essentially a handle for the result of an asynchronous message send. In our thread/actor protocol mapping, a synchronous Java invocation can be thought of as an asynchronous AmbientTalk message send that returns a future, on which the Java code immediately synchronises. Because of this immediate synchronisation, it has to be noted that the full power of futures is not really required: Java's standard `wait` and `notify` primitives suffice. However, in the next section we discuss how futures could help in achieving true symbiotic future-type message sends. Although our symbiosis hosts a trivial application of futures, to the best of our knowledge it has not before been used to synchronise between method invocations performed in different languages.

8 Research Status and Future Work

The AmbientTalk language is implemented as an interpreter on top of the Java Virtual Machine[3]. AmbientTalk is a research artifact, serving as a practical platform for experimentation, but lacking extensive performance optimisations. The interpreter runs on the Java Micro Edition platform and has been successfully deployed on PDAs connected via an ad hoc WiFi network. We are currently planning on applying the symbiosis by distributing a sequential Java application using AmbientTalk as the "distribution middleware".

With respect to the AmbientTalk/Java data mapping, one area of future work is the ability for AmbientTalk objects to "subclass" Java classes. This would allow AmbientTalk objects to function directly as Java anonymous inner classes without the need

[3] The language can be downloaded at `http://prog.vub.ac.be/amop/at/download`.

for manually written adaptor classes such as the `ATTimerTask` introduced in section 5.3, improving AmbientTalk's ability to integrate with existing Java frameworks.

With respect to the AmbientTalk/Java protocol mapping, we are planning on extending it with more elaborate support for future-type message sending. Although we have not discussed it in this paper, AmbientTalk supports asynchronous message sends that return futures. AmbientTalk's futures are based on the E language's non-blocking futures [8]: an event loop is never allowed to block, waiting for a future to become fulfilled. Instead, it is possible to register a listener on the future which is notified asynchronously when the future is fulfilled. Since version 1.5, Java also supports futures (as a library abstraction), but these are traditional futures that allow a thread to block, waiting for the future to become fulfilled. A logical next step in the symbiosis is to define a mapping between AmbientTalk's non-blocking and Java's blocking futures. This protocol mapping would fill a missing gap in the symbiosis: currently a Java invocation is either purely asynchronous or purely synchronous. If the symbiosis would support a future mapping (e.g. by detecting that the invoked Java method's return type is a subtype of `java.util.concurrent.Future`), Java invocations can be turned into future-type asynchronous AmbientTalk message sends.

9 Conclusions

We have described the AmbientTalk/Java linguistic symbiosis, which defines a protocol mapping between the event-driven concurrency model of AmbientTalk and the multithreaded concurrency model of Java. The motivation behind the symbiosis is that AmbientTalk can benefit from Java's vast number of existing components, while Java can benefit from AmbientTalk's high-level concurrent and distributed language features. The main problem to be tackled by the symbiosis is that Java's threading model should not violate the properties of the event-driven concurrency model (e.g. the prevention of race conditions on AmbientTalk objects).

The contribution of this paper is a protocol mapping between actors and threads which ensures that the concurrency properties of the actor-based model are preserved. To reconcile Java's synchronous method invocations with AmbientTalk's asynchronous message sends, we have distinguished three different kinds of symbiotic invocations: immediate invocations (performed by the actor's own thread representation), synchronous invocations (which require a Java thread to wait for an event loop) and asynchronous invocations (which treats Java method invocations on listeners as pure asynchronous message sends). We have described an application of each symbiotic invocation by means of concrete examples in our AmbientTalk/Java symbiosis.

Acknowledgments. The authors would like to thank Kris Gybels for his suggestions for improvement, in particular the structuring of the section on the actor/thread protocol mapping, and the anonymous referees for their helpful suggestions for improvement.

References

1. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented Programming in Ambienttalk. In Thomas, D., ed.: Proceedings of the 20th European Con-

ference on Object-oriented Programming (ECOOP). Volume 4067 of Lecture Notes in Computer Science., Springer (2006) 230–254
2. Dedecker, J., Van Belle, W.: Actors for mobile ad-hoc networks. In Yang, L., Guo, M., Gao, J., Jha, N., eds.: Embedded and Ubiquitous Computing. Volume 3207 of Lecture Notes in Computer Science., Springer-Verlag (2004) 482–494
3. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: Conference proceedings on Object-oriented Programming Systems, Languages and Applications, ACM Press (1986) 214–223
4. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
5. Lieberman, H.: Concurrent object-oriented programming in ACT 1. In Yonezawa, A., Tokoro, M., eds.: Object-Oriented Concurrent Programming. MIT Press (1987) 9–36
6. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press (1986) 258–268
7. Briot, J.P.: From objects to actors: study of a limited symbiosis in smalltalk-80. In: Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming, New York, NY, USA, ACM Press (1988) 69–72
8. Miller, M., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In Nicola, R.D., Sangiorgi, D., eds.: Symposium on Trustworthy Global Computing. Volume 3705 of LNCS., Springer (2005) 195–229
9. Gybels, K., Wuyts, R., Ducasse, S., D'Hondt, M.: Inter-language reflection: A conceptual model and its implementation. Computer Languages, Systems & Structures **32**(2-3) (2006) 109–124
10. Ungar, D., Chambers, C., Chang, B.W., Hölzle, U.: Organizing programs without classes. Lisp Symb. Comput. **4**(3) (1991) 223–242
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
12. Steyaert, P.: Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks. PhD thesis, Vrije Universiteit Brussel (1994)
13. Wuyts, R., Ducasse, S.: Symbiotic reflection between an object-oriented and a logic programming language. In: ECOOP 2001 International Workshop on MultiParadigm Programming with Object-Oriented Languages. (2001)
14. Achermann, F., Nierstrasz, O.: Applications = Components + Scripts – a tour of Piccola. Software Architectures and Component Technology (2001) 261–292
15. Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Proc. Joint Modular Languages Conference. Springer LNCS (2006)
16. Haller, P., Odersky, M.: Actors that Unify Threads and Events. In: International Conference on Coordination Models and Languages. Lecture Notes in Computer Science (LNCS) (2007)
17. Li, P., Zdancewic, S.: Combining events and threads for scalable network services. In: ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), ACM Press (2007)
18. Baker Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. In: Proceedings of Symposium on AI and Programming Languages. Volume 8 of ACM Sigplan Notices. (1977) 55–59
19. Liskov, B., Shrira, L.: 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, ACM Press (1988) 260–267
20. Caromel, D.: Towards a method of object-oriented concurrent programming. Communications of the ACM **36**(9) (1993) 90–102