Event Loop Coordination Using Meta-programming

Laure Philips, Dries Harnie, Kevin Pinte, Wolfgang De Meuter

Event Loop Coordination using Meta-Programming

Laure Philips*, Dries Harnie**, Kevin Pinte, and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{lphilips,dharnie,kpinte,wdmeuter}@vub.ac.be

Abstract. Event-based programming is used in different domains, ranging from user interface development to various distributed systems. Combining different event-based subsystems into one system forces the developer to manually coordinate the different event loops of these subsystems. This leads to a lot of excessive code and, in addition, some event loops are prey to lifecycle state changes. On mobile applications, for example, event loops can be shut down when memory runs low on the device. Current approaches take care of the communication problems between the different types of event loops, but become complex when trying to deal with lifecycle state changes. We propose a new coordination model, Elector, that allows two event loops to run separately, and introduce a novel kind of reference, called undead references. These references do not only allow communication between the event loops, but also handle lifecycle state changes in such a way that they do not influence other event loops.

Keywords: Event loops, Coordination model, Mobile platforms, Ambient-oriented programming

1 Introduction

In traditional programming the control flow of a program is determined by its structure. To allow the program to react upon input in the of form events, the event-based programming paradigm can be used. The programmer registers observers or event handlers for different types of events and the *event loop* is responsible for detecting events and dispatching it to the observers. This programming paradigm is popular for developing user interfaces, where the program reacts upon input from the user. Event-based programming proves to be useful in distributed programming, where the source of an event and the matching event handler can live on different devices.
When developing a larger software system, composed out of different event-based

* Funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).
** Prospective research for Brussels, Innoviris

subsystems, the programmer needs to manually coordinate these event loops. Because the event loops have different characteristics, take for example the start-up time, it is up to the programmer to take care of the mismatch between the event loops. Current systems only address this issue partially; they do not take into account the lifecycle of event loops. Some event loops are prey to lifecycle state changes, meaning that an event loop can be shut down at any moment in time and maybe restarted afterwards. For example, mobile applications, which often use event loops, can be shut down when running in the background. Typically, the programmer can react upon lifecycle state changes, e.g. pause a playing video when the application is no longer visible to the user. These changes have consequences: a destroyed event loop is no longer accessible and other event loops in the system thus must be aware of the lifecycle state of that event loop.
In this paper, we introduce a coordination model called "Elector", which allows different types of event loops to be coordinated. The model enables programmers to concentrate on the logic of the program, instead of writing glue code for the event loops. The main concept behind Elector are its undead references, which transparently handle these lifecycle problems. We contribute the design and implementation of Elector, as a framework for the AmbientTalk language. We also provide a validation of our approach by comparing the code complexity of different versions of a representative application. Finally, we present a validation of Elector's performance measurements.
This paper is organised as follows: we first discuss related work in section 2 after which we introduce a motivating example (section 3). Afterwards we discuss the problems that arise when implementing this case study in section 4. In section 5 we present "Elector", together with a concrete implementation for Android and AmbientTalk. Section 6 evaluates the Elector model by comparing the different implementations of our motivating example. Finally we conclude this paper and present future work.

2 Related Work

In this section we discuss related work: other models that encapsulate one of the event loops and component-based software architectures, which are tailored towards systems with different sub-components.

2.1 Event Loop Encapsulation

Often when integrating different subsystems into a new and larger system, one ends up with excessive code size. This is because the subsystems make certain assumptions about the system in which they are used [6]. For example, when combining different event-based systems that are not compatible with each other, the programmer must adapt one or more event loops.
The models we discuss allow the coordination of event loops, by encapsulating their own native event loop inside an external one and allowing the native event loop to handle its events at a regular basis. As a consequence, when an event

handler does not end immediately or not at all, the *entire* application is blocked and not only one of the subparts. These models solve the communication issues related to combining event loops, but do not take into account the possible lifecycle state changes of these event loops.

POE (Perl Object Environment) [1] is an event-based Perl framework for reactive systems, cooperative multitasking and network applications. POE provides bridges for other event loops that normally need complete control, for example GTK. Such a bridge between an external event loop and the POE event loop runs the external event loop and uses timer functions to allow the POE event loop to handle its events periodically.

Tcl (Tool Command Language) [11] is a cross platform programming language that can be used in different domains: web applications, desktop applications, etc. The Tcl event loops provides mechanisms that allow the programmer to have a more fine-grained control over the event mechanism. This way, one can embed a Tcl event loop inside applications that have their own event loop.

2.2 Component-based Software Architectures

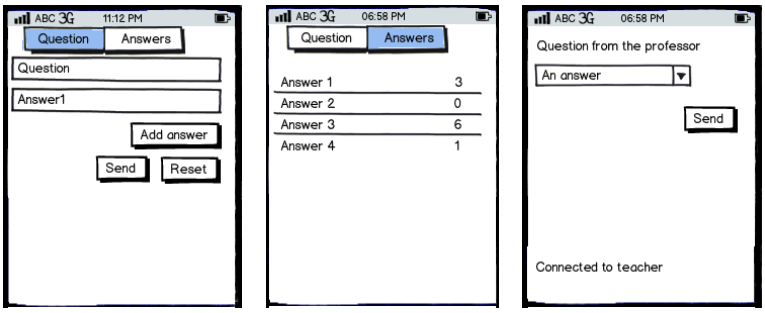
ROS (Robot Operating System) [8] uses topics to exchange messages between nodes in a publish/subscribe manner. More concretely, nodes subscribe to a certain topic, while other nodes publish data on this relevant topic. ROS allows newly connected nodes to subscribe dynamically. While this decouples the subscribers and publishers, the nodes need to be running at the same time to send data to each other. This means that ROS supports decoupling in space , because it decouples the publishers and subscribers, but not decoupling in time, because the nodes need to be active at the same time in order to communicate [10]. This is necessary when taking into account that components in a system can have a different lifetime. In the case of event loops, Elector does not require one event loop to wait for the other event loop to be (re)started. Elector allows event loops to send messages to an event loop that is not available upon the time of sending, but guarantees that the message will be delivered when that event loop becomes available.

Java Beans [5] are reusable components that can be composed using a visual composition tool. It is made for reusability by supporting persistence, introspection, customisation through property editors, etc. Beans communicate through events, where a bean can be the source or the target of the event. Beans register their interest as a listener at another bean. This also reduces the inter-component coupling, but they are not decoupled in time, so beans need to be running at the same time to communicate.

3 Motivating Example

In this section we introduce an example scenario, which is used to abstract the problems that arise when combining event loops. Our case study is a clicker

application, a so-called personal response system that is used as a communication system during lectures. Every student receives a remote control, also called zapper or clicker, that allows them to select an answer when the teacher asks a question. The software on the teacher's computer collects the answers and the results are shown to the class by representing them in a graph. Such a clicker application increases the interactivity and feedback during lectures [4], involving all students and ensuring anonymity.
Nowadays, all students are equipped with smart phones, tablets, ... Therefore we use these mobile devices as a clicker device and use a mobile network to communicate between the students and the teacher. Our clicker application can be started for a teacher or for students, so first of all the user has to choose one of these roles. The teacher can send questions to the students, together with possible answers (shown in figures 1(a) and 1(b)). The student receives these questions and can send a selected answer to the teacher (shown in figure 1(c)).



(a) Teacher: question (b) Teacher: answers (c) Student

Fig. 1. Android views of the clicker application

We will implement this application on the Android mobile platform using the AmbientTalk programming language [10]. In the following sections we introduce AmbientTalk and Android and how they use event loops.

3.1 AmbientTalk

AmbientTalk [3, 10] is a distributed programming language that is designed to solve the typical problems in mobile ad-hoc networks. A mobile ad-hoc network has no central infrastructure and the mobile devices that communicate with each other can move out of range. Moreover, the communicating parties can reconnect after a disconnection. AmbientTalk is based on the actor model [2], where every actor is represented by an event loop that communicates with the other event loops. AmbientTalk's communicating event loop model is tailored towards mobile ad-hoc networks: every event loop is independent, maintains its own state and communicates with other event loops by sending messages, thus events, to it. Moreover, network events like the discovery of actors are also handled by the event loop.

3.2 Android

An Android application is typically composed of different activities, where each activity represents the screen the user sees. An activity is the most important component of an Android application. Because the user can only see one screen at the time, only one activity can be active or in the foreground. Figure 1 shows the different screens the application consists of, each thus implemented by an activity.

The application components that run in the background can be either still running, but not visible, or can be destroyed. The programmer thus needs to take into account that an application running in the background can be destroyed, meaning all its state is lost. This is not only so for the Android platform, it is a characteristic shared by other mobile platforms like Apple iOS, BlackBerry OS, Windows Phone, etc. These mobile platforms define at least three lifecycle states for applications: active or running in the foreground, suspended or running the background and not running or destroyed. Thus, mobile applications are subject to lifecycle state changes, which are caused by the system or by the user navigating between applications.

When the lifecycle changes, the programmer can react by using callback methods. When for example an application that plays videos is suspended, the "pause" callback method can pause the video. When this application is destroyed, the programmer can store which video the user was watching.

If we take for example the student activity of the clicker application, we need to take into account that the reference to the corresponding AmbientTalk actor can be lost when the activity is restarted. Both event loops establish these references by implementing a registering method. In case the user is a student, the student activity starts the student actor at the beginning of the application. The actor retrieves a reference to the activity by calling its `registerAT` method and passing a reference to the student actor (shown in listing 1.2). Inside this registering method, the activity saves this reference to the actor and returns a reference to itself, as can be seen on line 1 of listing 1.1. On line 5 we see a simplified example of an event listener that gets called when the "send" button gets clicked. As can be seen on line 7, we need to check if the reference to the student actor is still valid. When the activity is restarted, we cannot restart the student actor, because the previous started one is still running.

```
1 StudentActivity registerAT(Student s) {
2   StudentActivity.student = s;
3   return this;
4 }
5 class Listener {
6  void onClick(View v) {
7    if (student != null) {
8    // send answer to teacher
9 }}}
```

Listing 1.1. Registering an actor (Java)

```
1 def gui := Android.registerAT(self)
```

Listing 1.2. Registering an activity (AmbientTalk)

Each application has a main thread, also called the UI thread, which cannot be blocked because it is responsible for processing lifecycle and user interaction events. Only the UI thread can alter view components, hence its name. When an event handler has a long running task to perform, it is up to the programmer to perform this task on a background or worker thread. As mentioned, these worker threads cannot update the user interface of the activity. To solve this, background threads have to use the `runOnUIThread` method for this purpose.

4 Problems

Using the building components of the AmbientTalk language and the Android platform we implemented a first version of the clicker application. The language integration is solved because AmbientTalk is implemented in Java (Android applications are implemented in Java) and the languages live in symbiosis with each other [9]. Therefore our solution does not take language integration into account, but focuses on the event loops. Several state-of-the art mechanisms exist to let code written in different languages interoperate, for example Mono, a framework for building cross-platform applications, integrates Java with languages of the .NET framework.
We give an overview of the problems that are encountered when combining different types of event loops. We can categorise these problems into lifecycle and communication problems. Lifecycle problems are caused by the different lifecycle of the event loops, while communication problems arise when the event loops send messages to one another.

Lifecycle Problems

Different startup time. When using two different event loops, chances are small that they have the same startup time. Because of this, one of the event loops needs to wait before communication between the two event loops can start. Therefore, it is up to the event loop that started later to initialise the communication, or to let the other event loop know it has started. When programming an application on Android using AmbientTalk, it is the task of the programmer to manually start up AmbientTalk and evaluate AmbientTalk actors. The mutual discovery is hard-coded by implementing registering methods in every event loop to establish references, as showed earlier in section 3.2.

Lifecycle state changes. Event loops that suffer from lifecycle state changes can be destroyed, which means that events or messages sent to them are not processed. For instance, Android activities can be killed when the user navigates to another activity or when memory is low. But in our clicker application we need to take into account that actors and activities refer to each other. When an activity is killed, all state is lost, including the reference to the actor and maybe the AmbientTalk interpreter. When the activity is restarted, the programmer needs to check if AmbientTalk is still running,

if the actors are still alive, etc. If so, the programmer must establish a way to reconnect the new instance of the activity and the actor. If not, the actor needs to be restarted. There is currently no trivial way to handle this problem and the programmer must implement ad-hoc solutions for handling these lifecycle state changes.

References can become invalid. Because of the lifecycle state changes, the references to an event loop can become invalid when it is destroyed. The other event loop does not know the reference has become invalid. In the case of AmbientTalk and Android event loops, we have seen that the references between them cannot be restored, when the activity is recreated. A restarted activity is actually a new instance of that activity, so upon creation of an activity, we need a way to make actors point to the newly started activity. Communication sent to a destroyed activity is lost, which brings us to the communication problems.

Communication Problems

Communication can be lost. This problem can arise in two different situations; when both event loops are not started yet, or when a reference to an event loop has become invalid. When a reference to an event loop is invalid, all messages sent to it are lost. When one of the event loops has not been started yet, the already started event loop has to wait before it can start to communicate with the other event loop. This is a consequence of the first problem. In case of Android and AmbientTalk event loops this means that Android's activities must wait when all the actors are started before they can send messages to it. Actors on their turn must be careful when sending messages to an activity, because it can be destroyed.

It is up to the programmer to establish a way for these event loops to communicate and handle each others lifecycle state changes. This leads to a lot of boilerplate code, which can be avoided as we discuss in the next section.

5 Elector

In this section we introduce the Elector (an acronym of Event Loop Coordination) model together with its implementation for the Android and AmbientTalk event loops. In Elector, references between different event loops are managed by undead references, which revive when a new event loop of the same type is started, hence their naming. On top of that, Elector wraps one of the event loops inside an event loop of the other type. The wrapped and wrapping event loop share the same lifecycle: when the wrapped one is killed, the wrapping is killed too. When the wrapped event loop becomes available again, so is its wrapper. Moreover, the wrapping event loop is the only one that has access to the wrapped event loop. The wrapping event loop thus routes incoming events directly to the wrapped event loop. Concretely, the wrapped and wrapping event loop refer directly to each other. Because they share the same lifecycle, this does not introduce the problems discussed before.

5.1 Model

Figure 2 shows how two different event loops are coordinated in Elector. We used AmbientTalk and Android event loops, but the model can be used for other types of event loops as well. Elector wraps the Android event loop inside an AmbientTalk event loop, which we call from now on the *wrapped* and *wrapping* event loops respectively. As we can see, the wrapping event loop refers to the

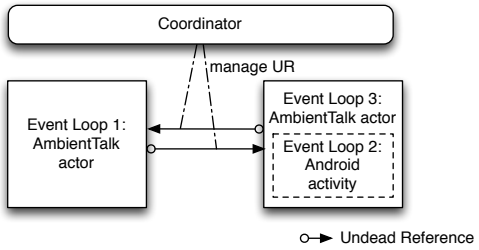


Fig. 2. Elector: the basic idea

other event loop of the same type by means of these undead references. The *coordinator* is responsible for managing the undead references and immediately returns a reference when asked for, even if the other event loops has not started yet. Undead references are the main concept of the model and they have the following characteristics:

Return immediately. When the coordinator is asked for an undead reference to a certain event loop, it returns one immediately, even when the referred event loop has not started yet. This way, event loops don't have to wait for the initialisation of the other event loop. In our example, the student actor retrieves an undead reference to the wrapping actor of the student activity, even when that activity is not started yet.

Remain accessible. Elector hides the state changes of an event loop behind undead references. Event loops that have an undead reference to an event loop that is hampered by lifecycle state changes, are not informed of this change and can keep communicating with the event loop as if it is still available. The student actor for example can continue using the undead reference it retrieved to the student activity, even when the activity is destroyed.

Rebind automatically. When an event loop is killed and subsequently restarted by the system, it is actually a new instance of that event loop. Undead references will rebind automatically to the new event loop. For example, after restarting the student activity all undead references to it transparently rebind with the new instance.

Buffer communication. Undead references route the communication to the actual event loop. When that event loop is inaccessible, the communication to it is buffered. When the event loop returns to a state that it can process messages, all the buffered ones are sent to it in the same order they were

received. In case of the clicker example, all messages sent to a wrapping actor for updating the user interface (e.g. displaying a question from the teacher) are sent when the wrapping actor and its wrapped activity are restarted.

By wrapping an event loop inside an event loop of another type, we can make use of the communication abstractions of that type of event loops. This results in an asynchronous way of communication between the event loops, which allows both of them to keep processing events, which is not the case of the models we discussed in section 2.
The coordinator creates the undead references, keeps track of them and also discovers all the event loops in the system. When an event loop is killed, the coordinator is informed and all undead references to that event loop start to buffer communication until a replacement event loop of the same type becomes available again. As we will see in the instantiation of Elector for Android and AmbientTalk, the coordinator should provide some guarantees that it is less likely to be killed. In our implementation, the coordinator is only killed when the whole application is destroyed, and thus all its subcomponents too.

5.2 Instantiation for Android and AmbientTalk

We implemented Elector for the Android and AmbientTalk event loops, where we choose to wrap the Android event loop inside an AmbientTalk event loop, being an AmbientTalk actor. There are several reasons for this choice: first of all, we can use the AmbientTalk discovery mechanism, which we discuss further on. Secondly, AmbientTalk allows event loops to communicate in an asynchronous and non-blocking way. Finally, AmbientTalk event loops don't suffer lifecycle state changes as much as Android event loops.

Coordinator The coordinator is in this case an AmbientTalk actor that discovers all wrapping actors, together with an Android service that orchestrates all the activities of the application. In our case, the activities of the application connect with the service. When the state of an activity changes, it must inform the service of this state change. Other than keeping track of the lifecycle of the activities, we use this service to automatically start an AmbientTalk interpreter in the background and to load code in this interpreter. Each Android activity thus has an associated AmbientTalk actor, and the programmer has to follow a naming convention in order for the evaluation of the actors to be done transparently.
When an Android application uses an Android service, it retrieves a higher priority and it is less likely that the application is destroyed. Should it happen that the service is killed under extreme memory pressure, the AmbientTalk interpreter is destroyed too, together with the coordinator actor and all other running actors. When the service is restarted, the AmbientTalk interpreter is restarted too and the required actors will be re-evaluated.

For the discovery we use AmbientTalk's discovery mechanism, as illustrated in the following code snippets.

```
1 def CoordinatorActor := actor: {
2   whenever: Activity discovered: { |e|
3     // retrieve tag, manage references
4    };
5   Android.coordinator = self }
```

Listing 1.3. Discovery of an actor

```
1 deftype StudentGUI <: Activity;
2 def remoteInterface := object: {
3   // behaviour
4 };
5 export: remoteInterface as: StudentGUI
```

Listing 1.4. Publishing an actor

On line 2 of listing 1.3 we see how the coordinator actor discovers all actors that are categorised in the network as (subtypes of) the Activity tag. The construct `whenever:discovered:` is used for this purpose. When an actor with that tag is discovered, the coordinator actor receives a *far reference* to the discovered actor. The other code snippet (1.4) shows the other side: inside a wrapping actor, we can export its behaviour in the network using `export:as:`. Tags are made with the `deftype` keyword. On line 1 we create a tag StudentGUI, that is a subtype of the Activity tag. The StudentGUI tag is defined by the wrapping actor of the activity StudentActivity. We now discuss how these wrapping actors behave.

Wrapping actors Wrapping actors share the same lifecycle as their wrapped activity. More concretely, the wrapping actor is taken offline when the corresponding activity is destroyed, and taken back online when the activity restarts. The coordinator can react upon these network events and inform all undead references to that particular wrapping actor.
Next to sharing the same lifecycle with its Android activity, the wrapping actor communicates with that activity and also allows us to write most of the user interface code inside these wrapping actors instead of the activities. The communication between a wrapping actor and its wrapped activity can easily be achieved because of the symbiosis [9] between the AmbientTalk and Java language. When executing a method call inside an AmbientTalk actor, this method gets executed on the thread of that actor. Recall from section 3.2 that only the UI thread may update Android views and as a consequence we cannot directly update the view from AmbientTalk. Therefore, we introduce a new kind of actor: the UI Thread actor. This actor is responsible for the communication between actors and its activities, by allowing actors to post Runnables on the Android event loop. The wrapping actors can *ship off* their messages to this actor, which guarantees that they are executed on the right thread. The following code shows how the student's wrapping actor defines a method that shows whether the student is currently connected to the teacher.

```
1 def teacherStatus(status) {
2   UIThreadActor←runOnUIThread(getTypeTag(), "connection_state", runnable: {
3     def run(v) {  // method to be executed on UI Thread
4       v.setText(status);    // set the text
5       v.setVisibility(v.VISIBLE); // show the text view
6  }})}
```

This code sends an asynchronous message to the UIThreadActor, denoted by the left arrow. The first argument of `runOnUIThread` (line 2) message selects the correct Android activity. As a second argument, we pass the name of the view we want to alter and lastly, we pass the AmbientTalk equivalent of a Java Runnable, denoted by the `runnable` keyword. It defines a method `run` on line 3, that takes the Android view as an argument. This way of changing Android views is only the first step: we discuss an improvement in the next section.

Undead references The final component of Elector are the undead references between an actor and a wrapping actor. AmbientTalk actors ask the coordinator for an undead reference to a wrapping actor using a tag, e.g. StudentGUI. The first time, the coordinator object makes a new one. When another actor already asked for an undead reference to that wrapping actor, the coordinator does not need to make a new reference, but reuses it. Figure 3 shows how an undead reference switches between two states: "forwarding" indicates the corresponding wrapping actor is available and in the "buffering" state the undead reference starts buffering all messages sent to it. Since the coordinator is informed of the availability of the wrapping actor, it is the task of the coordinator to switch the undead references between these states.

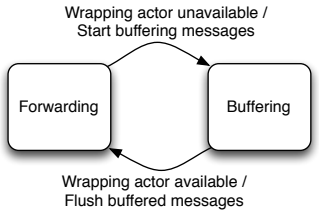


Fig. 3. State diagram for an undead reference

An undead reference is entirely programmed using AmbientTalk's reflective layer [7]. This layer allows us to catch asynchronous method calls on this reference and decide whether they need to be buffered (in case the corresponding wrapping actor is not available) or can be sent to the wrapping actor.
For example, the student actor can alter the text of the status view of the previous code snippet to "Not connected to teacher" when the student actor is started in the following way:

```
1 import /.at.android.undead_references;
2 deftype StudentGUI; // tag we want to retrieve
3 def guiRef := undeadRef(StudentGUI);   // ask undead reference from coordinator
4 guiRef←teacherStatus("Not connected to teacher");  // send asynchronous message
```

On line 4 we send an asynchronous message to the undead reference we retrieved on line 3. When the wrapping actor is already started, the message is immediately

forwarded to the wrapping actor. If not, the message is buffered and guaranteed to be delivered when the wrapping actor becomes available (again).

6 Preliminary Results

In this section we show the merits of the Elector model by comparing the ad-hoc implementation of the clicker application and the versions using the Elector model. But first of all, we validate the implementation of the Elector model by showing how it offers a base for a more polished version, adding more constructs that make it easier for the programmer.

6.1 Specific improvements to Elector for Android and AmbientTalk

The implementation of the Elector model as presented in the previous section is a direct translation of the general model. This implementation solves all the problems that arise when combining two different event loops. Because the user interface code has shifted to the wrapping actors, the Android part of the application contains less code. Therefore, we introduce new constructs that ease the writing of the wrapping actors.
First of all, we introduce a `run:` construct that takes a block of code as an argument. Behind the scenes, this code is sent to the Android UI thread, hiding the UI Thread actor from the programmer. Similarly, the `listener:` construct can be used to create an event handler for Android views.

Most of the UI code for an Android application are operations like setText, setColor, setVisibility,... on an Android view. These methods alter the user interface, thus they must be executed on the UI thread of the application. We introduce the `getView` method that returns a reference to an Android View object. All setters on this reference are automatically forwarded to the UI Thread actor. The following code snippet shows how `getView` can be used by the student wrapping actor to display a question and possible answers from the teacher.

```
1 import jlobby.android.view:
2 def askQuestion(question, answers) {
3   def question_v := getView("question");
4   def previous := question_v.getText(); // retrieve current text of view
5   question_v.setText(question); // alter text
6   question_v.setVisibility(View.VISIBLE); // show the text
7 }
```

This way, the programmer can get attributes of the view and alter them inside AmbientTalk, without worrying about thread-safety.
The final extension is the support for futures [3]. AmbientTalk's event loops communicate in a non-blocking, asynchronous way, meaning that an asynchronous message send between event loops immediately returns. AmbientTalk's futures are placeholders for the actual result of a message execution. When the actual result is computed, the future is resolved and other actors can receive this result

by installing an event handler on that future.
A common task in applications is to ask the user for some kind of input, and this is where futures are helpful. We extended Elector in such a way that futures can be resolved inside the code of a `run:` or `listener:` construct. This way, an AmbientTalk future can be resolved when e.g. a button is clicked. Vice versa, a text view can be linked to a future, filling in the view when the future becomes resolved.

6.2 Comparison of the clicker applications

We ended up with three versions of the clicker application: one without Elector, one that uses Elector and a final one that uses the extensions from previous section. The final version of the application spends less code on the mismatching concerns and contains less code overall. The first clicker version contains 512 lines of code, the second one 430 lines and the final version 367 lines of code, which is a decrease of 28% between the first and final one.

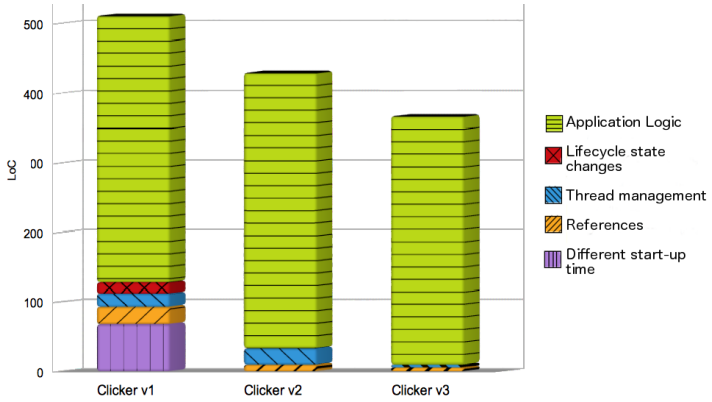


Fig. 4. Graph comparison for the first and final version of clicker

As we can see from the graphs in figure 4, Elector frees the programmer from dealing with the lifecycle of the event loops (lifecycle state changes and different start-up time). These issues are completely handled by Elector in the last two versions. Managing references and threads are reduced from 8,5% for the first version to 2,6% for the final one.
We also conducted memory and latency benchmarks. First, we measured the memory usage of the application at three different points: the choice between two roles, starting of the teacher, and starting of the student version. Starting the application uses 3.767 Mb for the first clicker and 3.671 Mb for the final one. For the teacher role we measured 5.112 Mb versus 9.293 Mb for the first and third version of Clicker respectively. For the student role these measurements are 4.82 Mb versus 7.384 Mb.
Secondly, we performed benchmarks to measure the latency of GUI updates in

the final version of clicker (the second and third version are almost identical). Therefore, we measured the latency between registering an event (clicking on a button) and the actual update of the GUI (in the Android UI code) (Scenario 1). We also measured the latency between registering the event and executing the `listener:` construct and from this point to the updating of the Android View (Scenario 2). From these experiments we obtained the following results (average and standard deviation of 200 experiments) in milliseconds:

	Scenario 1	Scenario 2	
		click → listener	listener → update
AVG +− STDEV	285.8 +− 39.73	4.55 +− 2.28	278.29 +− 47.57

Elector is a proof-of-concept implementation as a framework for AmbientTalk which we use to illustrate that our model eases the coordination of different event loops. When absolute performance is a necessity, Elector could be implemented in the AmbientTalk interpreter itself.
We can conclude that Elector had an impact on the clicker application: the programmer is freed from manually coordinating the event loops. As a side-effect, this has an impact on the code size and complexity, but moreover, more of the code now focuses on the logic of the program. This initial result is promising, but more studies are needed to further gauge the impact of Elector.

7 Conclusion and Future Work

We introduced a coordination model called Elector, that tackles the problems that arise when coordinating different event loops. Elector is targeted towards event loops that suffer from lifecycle changes (e.g. the event loops can be killed and restarted) but the model is suited for all sorts of event loops. In contrast to existing libraries, Elector allows the event loops to run separately, instead of letting one event loop take control over the other. Elector also does not change the source code of the event loops, but provides a bridge between them.
The main component of the Elector model are its undead references, the other components of the model manage these references. The undead references solve all the problems encountered when combining different event loops, namely the lifecycle and communication problems.
Using the implementation of the Elector model for the Android and AmbientTalk event loops, we evaluated the model by implementing a concrete case study: a clicker application. We compared the different versions of that application and concluded that Elector relieves the programmer from manually coordinating the event loops.
The Elector implementation is targeted towards event loops on one device, but we could extend Elector with *distributed undead references*. This way, a teacher could for example first demonstrate how the clicker application works by retrieving an undead reference to the user interfaces of the students. The other way around is also possible: several people can retrieve an undead reference to one

and the same user interface, e.g. for a brainstorming session.
An important task of the undead references of Elector is to buffer communication between event loops when one of them is not available (yet). The current implementation of Elector does not take duplication of messages into account, but we could easily support *smart buffering*. In order to achieve this, an annotation could be used, that allows the programmer to define the behaviour of messages in the buffer. So not only does Elector solve the problems that arise when combining different types of event loops, its undead references can be applied in other domains as well.

References

1. The POE Cookbook, Januari 2013. `http://poe.perl.org/?POE_Cookbook`
2. Gul Agha. Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA, 1986
3. Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Ambient-Oriented Programming in AmbientTalk. In In Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP), pages 230254. Springer, 2006
4. Ray dInverno, Hugh Davis, and Su White. Using a Personal Response System for Promoting Student Interaction. Teaching Mathematics and its applications, 22(4):163169, 2003
5. Robert Englander. Developing Java beans. OReilly & Associates, Inc., Sebastopol, CA, USA, 1997
6. David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch, or, Why its hard to build systems out of existing parts. In Proceedings of the 17th International Conference on Software Engineering, pages 179185, Seattle, Washington, April 1995
7. Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. Mirages: behavioral intercession in a mirror-based architecture. In Proceedings of the 2007 symposium on Dynamic languages, DLS 07, pages 89100, New York, NY, USA, 2007. ACM
8. Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In ICRA Workshop on Open Source Software, 2009.
9. Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic Symbiosis between Event Loop Actors and Threads. Computer Languages, Systems & Structures, 35(1):8098, April 2009
10. Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In Proceedings of the XXVI International Conference of the Chilean Society of Computer Science, SCCC 07, pages 312, Washington, DC, USA, 2007. IEEE Computer Society
11. Brent B. Welch. Practical programming in Tcl and TK (2. ed.). Prentice Hall, 1997