Electronic Communications of the EASST
Volume 11 (2008)

Proceedings of the
First International DisCoTec Workshop on
Context-aware Adaptation Mechanisms for
Pervasive and Ubiquitous Services
(CAMPUS 2008)

Reactive Context-Aware Programming

Stijn Mostinckx, Andoni Lombide Carreton, Wolfgang De Meuter

13 pages

Guest Editors: Romain Rouvoy, Mauro Caporuscio, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Reactive Context-Aware Programming

Stijn Mostinckx, Andoni Lombide Carreton, Wolfgang De Meuter

Programming Technology Lab
Vrije Universiteit Brussel

Abstract: Using state of the art tools, context-aware applications are notified of relevant changes in their environment through event handlers which are triggered by dedicated middleware. The events signalled by the middleware should percolate through the entire application, requiring a carefully crafted network of observers combined with complex synchronization code to address the inherent concurrency issues. This paper proposes the adoption of reactive programming techniques to bridge the gap between the event-driven middleware and the application.

Keywords: Fact Spaces, Reactive Programming, AmbientTalk

1 Introduction

The rapidly increasing capabilities of popular mobile computing devices such as cellular phones and PDAs herald a future where computing power has truly become ubiquitous. Using a present-day cellular phone, a user can interact with other proximate devices using e.g. Bluetooth or Wireless Ethernet. The introduction of near-field communication (NFC) additionally provides cellular phones a means to interact with disposable computers (such as passive and active RFiD tags) that can be attached to consumer goods.

Paralleling the hardware evolutions, various middleware approaches have been proposed to ease the development of context-aware applications which reap the benefits of being surrounded by a cloud of small interconnected devices. Certain approaches focus primarily on providing reusable abstractions to interact with sensors (e.g. the CONTEXT TOOLKIT [DSA01]), while other middleware focusses on the ability to communicate in networks with a fluid topology and transient connectivity. In the latter category, we encounter coordination languages such as LIME [MPR01] as well as publish/subscribe middleware such as STEAM [MC03].

Context-aware applications can respond to changes in their environment by registering event handlers with one of these middleware systems. Such an event handler is triggered whenever the middleware detects a relevant change in the current context. Relying on explicit event handlers has a number of severe consequences on the design of the context-aware application, which we describe in the remainder of this section. In this paper, we propose using reactive programming techniques to deal with context change events. We illustrate by means of a concrete case study (described in section 2) which abstractions are required to be able to write context-aware applications without resorting to an explicit event-driven style. An overview of these abstractions is provided in section 3. Before concluding, section 4 discusses the overall architecture of the proposed reactive context-aware programming system and highlights relations to related work.

1.1 Motivation: Impact of Event Handlers on Application Design

Relying on explicit event handlers to notify the application of changes to its context introduces a number of problems. A first important issue is that the event handlers are triggered by the middleware, which implies that they may be executed concurrently with the application code. In a programming language with shared state concurrency (such as Java), this implies that the application programmer is responsible to avoid race conditions on the application's state. In other words, if the event handler accesses application data structures, all accesses in the application code (including those in event handlers) should be explicitly protected with locks.

Another corollary of the fact that the middleware triggers the event handlers is that performing long-lasting computations in the event handler may render the middleware unresponsive. A well-known example of this restriction is found in the documentation of the Java Swing GUI framework, which recommends that event handlers should finish as soon as possible. This particular recommendation provides further encouragement for a programming style where event handlers rely on side effects to signal context changes to the application.

When context changes are signalled by modifying shared variables, it is the responsibility of the application programmer to manually keep track of how the different pieces of application code depend on the state of these variables. This implies keeping an explicit network of computations to be notified when changes to these variables are observed.

Having observed these three problems, we propose the use of reactive programming techniques to establish an expressive binding between context-aware applications and the underlying middleware which avoids these problems altogether. This proposal is based on our experience combining CRIME, a middleware implementing the Fact Space model [MSP+07] with AMBIENTTALK, a dynamically typed, object-oriented, concurrent language [VMG+07].

2 Case Study

This section presents the concrete case study we have performed to uncover how to establish an expressive binding between a context-aware application and the underlying middleware in such a way that the event-driven style employed by the middleware does not percolate through the entire design of the application. As the context-aware middleware, we have opted for CRIME, an implementation of the Fact Space coordination model [MSP+07]. This model provides applications with a logic coordination language to reason about information and services provided by nearby devices. While the adoption of a logic coordination language is novel, the binding between CRIME and the application is established by means of explicit event handlers to be supplied by the application programmer. In this regard, the Fact Space model is similar to conventional context-aware middleware such as the CONTEXT TOOLKIT [DSA01], publish/subscribe middleware such as STEAM [MC03] or coordination languages such as LIME [MPR01].

In our case study, we have established a binding between CRIME and AMBIENTTALK, an experimental object-oriented language which was explicitly conceived as a language laboratory [MVTT07]. Due to its conception as a language laboratory, AMBIENTTALK provides many of the required tools to allow a natural embedding of a domain-specific language (such as the logic coordination language offered by CRIME).

The remainder of this section describes the concrete case study we have developed in more detail. First, we provide a brief overview of CRIME, its federated fact space in which applications can publish context information and its logic coordination language which allows triggering application-level reactions to context changes. Subsequently, we provide a bird's eye overview of AMBIENTTALK and we conclude this section by describing a straightforward integration of CRIME rules in AMBIENTTALK which relies on explicit event handlers.

2.1 Fact Space Model

The Fact Space model [MSP+07] is a coordination model which provides applications with a *federated fact space*: a distributed knowledge base containing logic facts which are implicitly made available for all devices within reach. Consequently, when an application asserts a fact in the federated fact space, other devices that are within earshot can see the fact and react to its appearance. To react to the appearance of facts in the federated fact space, applications specify *logic rules*. These logic rules describe the causal relation between (a combination of) context observations and the application's reaction. In summary, the Fact Space model combines the notion of a federated fact space with a logic coordination language. In the remainder of this section we will discuss each of these components in slightly more detail.

2.1.1 Federated Fact Space

The Fact Space model equips every application with a private fact space in which application-specific facts can be stored. Additionally, applications may use an arbitrary amount of (named) interface fact spaces. Facts which are asserted in these interface fact spaces are transparently shared by the underlying implementation (CRIME) which aggregates all reachable interface fact spaces with the same name into a federated fact space. Hence, when interacting with its own interface fact space, the application can transparently access facts published by other devices in its immediate environment.

In essence, the distribution semantics of the federated fact space in the Fact Space model are largely similar to those of the federated tuple space of LIME [MPR01]. One notable difference is that whereas tuple spaces are based on the notion of a blackboard where messages can be published, read and removed, fact spaces are conceived as knowledge bases. This implies that in the Fact Space model both the assertion and retraction of a fact are meaningful events which may trigger reactions.

The fact that applications can respond to both the assertion and the retraction of facts is of particular importance when devices go in and out of earshot. For instance, when a new device is discovered, its interface fact space will be *engaged*. This implies that all its facts will be atomically and transparently *asserted* in the federated fact space. Conversely, when a device goes out of communication range, its interface fact space is said to be *disengaged* which implies that all facts that it contained are retracted. Since retraction is reified by the Fact Space model, applications which reacted to the appearance of a fact published in the disengaged space (possibly in combination with other facts) are provided with the opportunity to recover from the disappearance of that fact. This mechanism provides the Fact Space model with fine-grained support for the handling of disconnections [MSP+07].

2.1.2 Logic Coordination Language

The Fact Space model provides applications with a logic coordination language to cherry pick the relevant context information from the federated fact space. For instance, consider a cellular phone that wants to adapt its sound profile according to its current location. First of all, such an application would contain a set of preferences which detail which sound profile needs to be adopted in various room types. In the example application in listing 1, these preferences are encoded as facts published in the private fact space.

Listing 1: Facts and rules to change the profile of a cellular phone.

```
preference(meetingRoom, silent).
preference(office, default).

:Switch(?profile) :-
  location -> detected(myID, ?room),
  preference(?room, ?profile).
```

Subsequently, a CRIME rule is defined which has two preconditions. First of all, it requires that there is a `detected` fact in a public fact space (named `location`), which denotes that the cellular phone is located in a room of a particular kind. The kind of room will be bound to the logic variable `?room`. Additionally, a `preference` should be defined which determines the sound profile to be adopted for the given kind of room. When both conditions are met, the profile to adopt will be bound to the logic variable `?profile` and the rule will be triggered. When the rule is triggered, it will invoke the application-specific action `:Switch` to ensure the cellular phone adopts the correct sound profile.

Applications can supply application-level actions (such as `:Switch`) by providing a Java class which implements the `Action` interface. Such classes need to implement two methods: First of all, the `activate` method which is invoked whenever a rule is triggered which lists the action in its rule head. Second, the `deactivate` method which is invoked when one of the facts that caused the rule to trigger is retracted. In the example, the `deactivate` method can be used to make the cellular adopt a loud sound profile.

2.2 AmbientTalk

As part of our case study, we have allowed CRIME rules to be embedded into context-aware applications written in AMBIENTTALK [VMG+07]. Before discussing this embedding in detail, we provide a bird's eye overview of the AMBIENTTALK language, which inherits most of its standard language features from Self, Scheme, Smalltalk and E. 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. Inspired by Self, AMBIENTTALK features classless slot-based objects. The concurrency model of the language was adopted from the E programming language.

2.2.1 AmbientTalk Objects

AMBIENTTALK is a dynamically typed, object-based language. Computation is expressed in terms of objects sending messages to one another. Objects are not instantiated from classes.

Rather, they are either created *ex-nihilo* or by cloning and adapting existing objects. The code excerpt in listing 2 defines such an ex-nihilo object and binds it to a variable named `CellPhone`. This object serves as a prototypical cellular phone object and can be used to create clones, as shown on the last line. Every object understands the message `new`, which clones the receiver object and initializes the clone by invoking its `init` method with the arguments passed to `new`.

Listing 2: AMBIENTTALK definition of the prototypical behavior of a cellular phone.

```
def CellPhone  := object: {
  def inbox    := []; // defines a field 'inbox'
  def state    := profiles.withName("default");
  // this method serves as the "constructor"
  def init(aProfile) {
    inbox    := [];
    state    := aProfile;
  };
  // method to prepend incoming messages to the inbox
  def receive(aShortMessage) {
        state.signalMessageReceipt(aShortMessage);
    inbox := [aShortMessage] + inbox; };
}
def myPhone := CellPhone.new(profiles.withName("silent"));
```

The cellular object object has a `state` field which is used to determine the sound profile of the cellular phone. For instance, when a short message (SMS) is received, the state is consulted to signal the receipt of the message (by e.g. vibrating, blinking or playing a particular ring tone).

2.2.2 Concurrent Programming in AmbientTalk

In AMBIENTTALK, concurrency is not spawned by means of threads but rather by means of actors [Agh86]. AMBIENTTALK actors are not conceived as active objects, but rather as *communicating event loops* which encapsulate a suite of objects [MTS05]. Objects encapsulated by a single actor can communicate by either invoking one another's methods synchronously or by sending asynchronous messages to one another. Communication with objects encapsulated by another actor is only possible by sending them asynchronous messages.

When an asynchronous message is sent to an object, it is enqueued in the message queue of the actor that encapsulates that object. The actor's event loop dequeues such asynchronous messages one by one and invokes the corresponding method of the object denoted as the receiver of the message. By processing messages serially, race conditions on the mutable state of the encapsulated objects are avoided.

2.2.3 Embedding Languages in AmbientTalk

AMBIENTTALK is conceived as a language laboratory and hence provides mechanisms which make it straightforward to make both syntactic and semantic extensions to the language. For instance, built-in control structures (e.g. `while:do:`) and language constructs (e.g. `object:`) are not conceived to be special. Instead, they exploit the fact that AMBIENTTALK supports both canonical syntax and keyworded syntax for method definitions and message sends. Since language constructs can be defined as ordinary functions, albeit ones using keyworded syntax, new language constructs can be introduced in a natural way.

AMBIENTTALK provides support for block closures reminiscent of those in Self and Smalltalk. A block closure is an anonymous function object that encapsulates a piece of code and the bindings of lexically free variables and `self`. Block closures are used to represent *delayed* computations, such as the branches of an `if:then:else:` control structure. Block closures are constructed by means of the syntax {`|args| body`}, where the arguments are optional.

2.3 Embedding Crime into AmbientTalk

Having briefly introduced both CRIME and AMBIENTTALK, we now describe how CRIME rules can be embedded in a context-aware application written in AMBIENTTALK. Embedding the rules in the context-aware application allows directly passing an AMBIENTTALK object which will implement the application-specific action. As before, objects which are to be used as an application-specific action should implement an `activate` and a `deactivate` method. The `activate` method will be invoked when the rule is triggered, which the `deactivate` method allows responding to the retraction of facts which led to a previous activation.

Listing 3 defines a `Switch` object which implements both methods. Upon activation it modifies the sound profile of the cellular phone to correspond to the preference derived by the rule. When the object is signalled that the previously adopted profile is no longer relevant (e.g. because the user has changed location), it will adopt a loud sound profile.

Listing 3: Symbiotically changing the profile of a cellular phone.

```
def Switch := object: {
  def activate(p)   { myPhone.state:= profiles.withName(p); };
  def deactivate(p) { myPhone.state:= profiles.withName("loud"); }; };

publish: { preference(meetingRoom, silent) };
publish: { preference(office, default) };

trigger: Switch whenInContext: {
  public -> location(myID, ?room);
  preference(?room, ?p); };
```

Interacting with the underlying CRIME engine is achieved through the introduction of new dedicated language constructs. First of all, a `publish:` construct is provided which expects a block closure. The body expression of this closure will be interpreted as a CRIME fact which is published privately by the underlying engine[1]. Second, a `trigger:whenInContext:` construct is provided to register an AMBIENTTALK object as the application-level action to be performed whenever a set of context conditions (described in a block closure) are met.

Whenever an embedded rule is triggered, the CRIME engine will invoke the `activate` method of the AMBIENTTALK object that was designated to be the corresponding application-level action. However, rather than invoking the method directly on the AMBIENTTALK object, CRIME interacts with a dynamically generated proxy object, which implements the `Action` interface. This proxy object translates method invocations from a Java thread into asynchronous messages [VMD07]. This translation step ensures that context event handlers can never be run in parallel with application code. Moreover, since the proxy can determine that the thread of the reasoning engine is not awaiting a result, it can resume immediately [VMD07].

[1] Similarly a `publish:in:` construct is provided which allows publishing a fact in a named public fact space.

One remaining issue is that dependencies on the context-dependent part of the application need to be managed manually. Managing such dependencies requires the programmer to resort to registering observers and firing events manually. The next section discusses how this issue can be addressed in a structured fashion through the introduction of reactive programming techniques.

3 Reactive Context-Aware Programming

Reactive programming is a programming paradigm built around the notion of time-varying values (called dataflows or *signals*). Changes to the values of these signals are automatically propagated to a network of dependent computations. Reactive programs construct this network either by explicitly wiring signals, or implicitly by calling *lifted* functions which operate on signals rather than ordinary values [EH97, CK06].

Whether one can apply a lifted function on a signal depends on whether the signal is continuous, i.e. whether it has a value at each point in time. Continuous signals (called *behaviors*) form the crux of the reactive programming model: by applying lifted functions on them, a dependent behavior is constructed transparently. Subsequent changes of the behaviors that were passed as arguments to the lifted function will be propagated transparently to the dependent behavior, allowing its value to be recomputed.

Signals which only carry events at discrete points in time are called *event sources*. These need to be manipulated using explicit operations (e.g. filter, merge and map). Event sources provide a natural mechanism to interact with the real world, where events can be generated by input/output devices such as mice or keyboards. A contribution of this paper is that it maps the events generated by context-aware middleware onto event sources in a reactive program.

3.1 Reactive Programming in AmbientTalk

Since AMBIENTTALK is an object-oriented programming language, we have explored an object-oriented reactive programming style to ease the transition from context-aware middleware to application-level code. We adopt the embedding strategy proposed in FRTIME [CK06], where reactive programming concepts are embodied in a host language. This implies that evaluating method invocations can be used to implicitly construct a network of dependent computations.

Consider for instance a behavior that denotes the current temperature in the room (`tempBeh`)[2]. When we compare the temperature behavior to a constant (`tempBeh < 18`), the receiver of the < method is a behavior. Hence, the < method is implicitly lifted and a dependent behavior (which will alternate between true and false) is created. The initial value of the dependent behavior is computed by comparing the current value of `tempBeh` to 18. Whenever `tempBeh` changes, the dependent behavior will be recomputed as well.

Methods are not only lifted implicitly when their receiver is a behavior, but also when one or more behaviors are passed as arguments. For instance, if we invert the above comparison (`18 > tempBeh`), the receiver of the > method is a plain AmbientTalk object (the number 18). However, since the argument of the method is a behavior (`tempBeh`), the method will still be lifted and yield a dependent behavior which alternates between true and false.

[2] How to create such a behavior from a context query is illustrated in listing 5.

3.2 Mapping Context Events to Behaviors

Having introduced support for reactive programming in AMBIENTTALK, it is possible to consider a more expressive embedding of CRIME rules in the language. Since CRIME essentially fires events whenever an interesting context change occurs, it seems natural to provide a construct which installs a rule and returns an event source which may be used in the context-aware application. The `contextQuery:` construct does just that: it takes a block closure as an argument and installs a CRIME rule with the content of this block as its precondition. The head of the rule, which is supplied implicitly, is a built-in action which will emit an event to the event source that is returned by the construct.

As an example, consider a context-aware application which displays the current room temperature on a gauge user interface component. Listing 4 contains an excerpt of the required code to build such an application. The `contextQuery:` construct is used to create an event source which carries events containing the current room temperature. These events are AMBIENTTALK objects which define corresponding fields for all variables bound by the query, to wit `?room` and `?temperature`. To distinguish between events which signal activation and deactivation of the rule, events are tagged with the `Activated` or `Deactivated` type tag[3].

Listing 4: Representing the outcome of a context query as an event source.

```
def gauge   := GaugeWidget.new(...);

def tempSrc := contextQuery: {
  public -> location(myID, ?room);
  public -> temperature(?room, ?temperature); };

tempSrc.foreach: { | event |
  if: (is: event taggedAs: Activated) then: {
    gauge.setHeight(event.temperature);
  } else: { // deactivated
    gauge.setHeight(nil); } }
```

Listing 4 illustrates the fallacies of having to directly interact with event sources: the programmer still has to manually construct a network of dependent computations using dedicated operators (e.g. `foreach:` which is used in the example). As we have already mentioned earlier in this section, the chief expressive gain of reactive programming is obtained when applying lifted functions to behaviors. To appreciate the difference between both styles, consider the code in listing 5, which uses a behavior to represent the current temperature. This behavior is created using the `mostRecent:fromContext:` construct which extracts one of the variables bound in the query, to wit the `?temperature`.

Listing 5: Constructing a behavior from a context query.

```
def tempBeh := mostRecent: ?temperature fromContext: {
  public -> location(myID, ?room);
  public -> temperature(?room, ?temperature); };

gauge.setHeight(tempBeh);
```

[3] Type tags are introduced as a strictly nominal type system in AMBIENTTALK to allow for the classification of objects. They are best compared to empty ‘marker’ interfaces such as `Cloneable` and `Serializable` in Java.

Using such a behavior, the `setHeight` method needs to be invoked only once (as if to initialize the gauge). Since the argument is a behavior, the method is implicitly lifted, such that it will be invoked again whenever the temperature changes. In this setup, retraction is handled by the `mostRecent:fromContext:` construct by setting the value to `nil`.

3.3 Reactive Context-Aware Collections

When converting an event source into a behavior, keeping track of the last propagated event seems to be an intuitive solution. However, this particular type of conversion is rarely the most appropriate one in a context-aware application. Consider for example, context queries to detect nearby users, context-aware reminders to be delivered to you, or nearby products which are on your shopping list. Each of these queries conceptually returns a group of results which are not adequately represented by only keeping track of the last event fired by the middleware.

Modelling the results of such context queries is achieved through the introduction of *reactive context-aware collections*. These collections allow abstracting over the events fired by CRIME such that the programmer who can work directly with the results of the query in a collection. Since the collection is modelled as a behavior, updates to the collection are implicitly propagated through the application. Listing 6 illustrates how to construct such a collection using the `collect:fromContext:` language construct.

Listing 6: Constructing a reactive context-aware collection.

```
def nearbyBeh := collect: ?user fromContext: {
  public -> location(myID,   ?room);
  public -> location(?user, ?room);
  ?user  != myID; };
```

The example in listing 6 accumulates all bindings for the `?user` variable in the given query. The resulting context-aware collection hence contains all collocated users. Note that the accuracy of the reactive context-aware collections hinges on the fact that CRIME reifies both the assertion and retraction of information. Without proper support for fine-grained reactions to the retraction of facts, context-aware collections would grow monotonously. Since retraction is properly reified, results can be deleted from the collection such that it always provides an up-to-date view on the current context.

3.4 Summary

Having presented an overview of the binding we have developed between AMBIENTTALK and CRIME, the characteristics of a reactive context-aware programming system can be clearly identified. First of all, the programming language should provide support for *reactive programming*, such that it is possible to apply lifted methods to behaviors. This is essential to abstract over the underlying propagation of (context change) events in the system.

Secondly, the programming language should adopt an *event loop concurrency* model, which ensures that reactive computation (propagating events and performing the dependent computations) can never be performed in parallel with application code. Since race conditions are avoided, methods which perform side effects can be safely lifted to be applied on behaviors.

Thirdly, to cater to the fact that context queries may report more than a single result, we advocate the introduction of *reactive context-aware collections*. These collections provide the reactive program with a collection of valid results, which is assembled by interpreting the events sent by the underlying middleware.

Finally, to uphold the consistency of the reactive context-aware collections, the underlying middleware should not only reify the appearance of new information but also the *disappearance of information*. The latter is a cue for removing the corresponding result of the context query from the associated collection.

4 Discussion

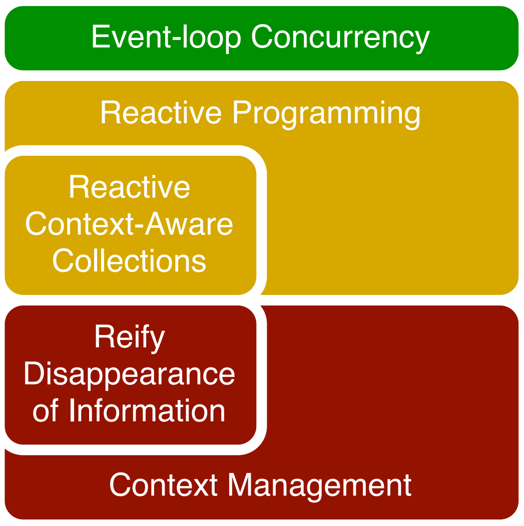
Event-loop Concurrency
Reactive Programming
Reactive Context-Aware Collections
Reify Disappearance of Information
Context Management

Figure 1: A Reactive Context-Aware Architecture

This paper proposes reactive context-aware programming, a paradigm which allows context-aware applications to abstract from the underlying events that result from interacting with the real world. Figure 1 provides an architectural overview of a reactive context-aware programming system. The system relies on the adoption of an event-loop concurrency model which ensures that concurrency within an event loop can be avoided. Underneath, a reactive programming system is provided to abstract over the events produced by the context management layer.

The introduction of reactive programming techniques avoids having to design context-aware applications using explicit event handlers. The use of reactive programming techniques to model context-aware application raises the level of abstraction at which the programmer reasons about the problem. The expressive power of using reactive programming has previously been illustrated in the context of user interface interaction [EH97]. In this field, application design was similarly sprinkled with event handlers, using side effects to communicate, and requiring manually managed dependencies.

When moving from user interface interaction to context-aware applications, an important observation is that the occurrence of a new event no longer automatically invalidates previous events. Many context-aware queries conceptually return a group of results, such as the collection of all collocated users. Reactive context-aware collections were proposed as a mechanism to capture the results of such queries, while still allowing for a reactive programming style. The idea of using context-aware collections to bridge the gap between context-aware applications and the underlying middleware was originally proposed by Payton et al. [PRJ04]. However, the collections they proposed could only be accessed using synchronous operations such as `get` and `remove`. Hence, to be notified of interesting context changes, applications have to poll the collection regularly. This pattern makes it cumbersome to develop applications which react to changes in their environment. The introduction of *reactive* context-aware collections reconciles the notion of context queries yielding multiple results with a reactive programming style.

When accumulating the results of a context query, being able to determine when a result is no longer valid is crucial. For this purpose, the underlying context management layer should reify not only the appearance of new information (which may add new results to the collection) but also the disappearance of information (which causes results to become invalid). CRIME is able to signal the disappearance of information since its forward chaining infrastructure intrinsically stores which (implicit) event handlers need to be notified when a particular fact is removed [MSP+07]. In spite of the fact that middleware approaches such as FACTS [TWS06], SENTIENT OBJECTS [BC04] and GAIA [RC03] provide a similar model based on distributed facts, rules and functions, these functions are only called when the corresponding rule is triggered by the appearance of new information. As such they cannot be directly used as a substrate on which a reactive context-aware programming system can be built.

5 Conclusion

Writing context-aware applications involves reacting to changes in the environment to adopt or fire the correct behavior. Using state of the art middleware, applications are required to register event handlers which will be triggered when context changes occur. Relying on explicit event handlers has effects which percolate throughout the entire design of the application, as access to shared data needs to be protected from race conditions and context dependencies in the program need to be encoded manually by registering observers.

In this paper we have advocated the use of reactive programming techniques to abstract from the propagation of events in a context-aware application. The adoption of reactive programming introduces some hurdles to overcome. First of all, classic reactive programming approaches introduce a second independent thread which propagates events and executes dependent computations. Since reactive programming is traditionally employed in a purely functional language, the introduction of a parallel process is not deemed harmful. In our case study, we have shown that it is possible to reconcile reactive programming with an imperative object-oriented language, provided that objects are properly encapsulated in an event loop. Adopting such an event loop concurrency model ensures that an object's methods are only invoked by the owning event loop and allows the safe interleaving of reactive computation with application behavior.

Much of the expressive gain when using reactive programming stems from the ability to apply lifted methods to behaviors, such that the result of the method will be recomputed implicitly whenever the value of the behavior changes. When mapping events generated by context-aware middleware to such behaviors, we observe that context queries often return a collection of interesting values. Therefore, we advocate in favor of the adoption of reactive context-sensitive collections which accumulate all relevant context information returned by a single context query. To keep these collections up to date, the context-aware middleware should have a mechanism to detect obsolete context information, such that it can be removed from the collection.

Based on these observations, we propose a model for reactive context-aware programming which entails 1) a context middleware with support to detect the invalidity of context information, 2) a programming language with an event loop concurrency model, 3) language-level support for reactive programming and 4) a library of reactive context-sensitive data structures which provide the programmer the necessary abstractions to model and react to context changes.

Acknowledgements: Stijn Mostinckx and Andoni Lombide Carreton are funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). The authors would like to thank Tom Van Cutsem and the anonymous reviewers for their insightful comments which benefitted the clarity of the paper.

Bibliography

[Agh86] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[BC04] G. Biegel, V. Cahill. A Framework for Developing Mobile, Context-aware Applications. In Tripathi et al. (eds.), *Proc. of the 2nd Int. Conf. on Pervasive Computing and Communications (PerCom)*. Pp. 361–365. IEEE Comp. Soc., Mar. 2004.

[CK06] G. Cooper, S. Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In Sestoft (ed.), *Programming Languages and Systems*. LECT NOTES COMPUT SC 3924, pp. 294–308. Springer Verlag, Mar. 2006.

[DSA01] A. K. Dey, D. Salber, G. D. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-aware Applications. *Human-Computer Interaction* 16(2–4):97–166, 2001.

[EH97] C. Elliott, P. Hudak. Functional Reactive Animation. In Tofte (ed.), *Proc. of the 2nd Int. Conf. on Functional Programming (ICFP)*. Pp. 263–273. ACM Press, 1997.

[MC03] R. Meier, V. Cahill. Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications. In Stefani et al. (eds.), *Proc. of the 4th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS)*. LECT NOTES COMPUT SC 2893, pp. 285–296. Springer Verlag, Nov. 2003.

[MPR01] A. L. Murphy, G. P. Picco, G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In Dasgupta and Zhao (eds.), *Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS)*. Pp. 524–536. IEEE Comp. Soc., Apr. 2001.

[MSP$^+$07] S. Mostinckx, C. Scholliers, E. Philips, C. Herzeel, W. De Meuter. Fact Spaces: Coordination in the Face of Disconnection. In Murphy and Vitek (eds.), *Proc. of the 9th Int. Conf. on Coordination Models and Languages (COORDINATION)*. LECT NOTES COMPUT SC 4467, pp. 268–285. Springer Verlag, June 2007.

[MTS05] M. Miller, E. D. Tribble, J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In Nicola and Sangiorgi (eds.), *Proc. of the Sym. on Trustworthy Global Computing (TGC)*. LECT NOTES COMPUT SC 3705, pp. 195–229. Springer Verlag, Apr. 2005.

[MVTT07] S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Tanter. Mirages: Behavioral Intercession in a Mirror-based Architecture. In Costanza and Hirshfeld (eds.), *Proc. of the Dynamic Languages Sym. (DLS)*. Pp. 222–248. ACM Press, Oct. 2007.

[PRJ04] J. Payton, G.-C. Roman, C. Julien. Context-Sensitive Data Structures Supporting Software Development in Ad Hoc Mobile Settings. In Choren et al. (eds.), *Proc. of the 3rd Int. Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS)*. Pp. 34–41. May 2004.

[RC03] A. Ranganathan, R. H. Campbell. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing* 7(6):353–364, Dec. 2003.

[TWS06] K. Terfloth, G. Wittenburg, J. H. Schiller. FACTS - A rule-based middleware architecture for wireless sensor networks. In Sanjoy and Venkatesh (eds.), *Proc. of the 1st Int. Conf. on Communication System Software and Middleware (COMSWARE)*. Pp. 1–8. IEEE Comp. Soc., Jan. 2006.

[VMD07] T. Van Cutsem, S. Mostinckx, W. De Meuter. Lymguistic Symbiosis between Actors and Threads. In Perrot and Demeyer (eds.), *Proc. of the Int. Conf. on Dynamic Languages (ICDL)*. ACM Press, Aug. 2007.

[VMG$^+$07] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. In Astudillo and Tanter (eds.), *Proc. of the XXVI Int. Conf. of the Chilean Computer Science Society (SCCC)*. Pp. 3–12. IEEE Comp. Soc., Nov. 2007.