Reasoning About Past Events in Context-Aware Middleware

Eline Philips, Christophe Scholliers,Charlotte Herzeel and Stijn Mostinckx
{ephilips, cfscholl, caherzee, smostinc}@vub.ac.be

Programming Technology Lab
Vrije Universiteit Brussel

1 Introduction

The hardware advances in networking technology of the past few decades have resulted in novel kinds of distributed systems, commonly referred to as *mobile ad hoc networks*. Such networks are populated by small, mobile handheld computers or cellular phones interconnected by highly volatile wireless communication links. Whereas the resources of these devices tend to be limited, their true strength stems from their ability to seamlessly integrate computing into our everyday life. A crucial factor to preserve this integration is that devices and the applications they host respond to the context in which they are situated.

The development of context-aware applications is widely supported by a large variety of frameworks such as JCAF [1], WildCAT [2], and Lime [6]. A common trait of these frameworks is that they cater for an event-driven programming style, where reactions are triggered as context events are fired. The one-to-one mapping between events and reactions requires that a reaction which is to be triggered when two context events occur simultaneously should install two separate event-handlers and hard-code the combination logic in the event handlers.

In previous work we have presented Crime, a logic coordination language which allows expressing reactions to be triggered upon the simultaneous occurrence of various context events [5]. By capturing these constraints in general-purpose rules, additional conditions can be imposed in a declarative fashion. Hence, Crime alleviates the need for imperative checks to be performed in body of the context event handlers.

On the other hand, we have also built Halo, an aspect-oriented extension to the Common Lisp Object System that supports reasoning about the execution history of a program in its logic-based aspect language, and thus enables expressing context-aware aspects [4].

In previous experiments, we have noted that although Crime allows one to reason about the current context, it can be equally interesting to reason about context events which were fired in the past. The next section provides an example scenario which illustrates a clear use case for this behaviour. This position paper proposes to integrate features of Halo into Crime as a basis to explore reasoning about past events in context-aware middleware.

2 Scenario

In previous work we have presented the following scenario as an example of a complex context-aware system whose behaviour can be expressed in a declarative fashion using Crime [5].

> Alice, Jim and Bob are students which share an apartment. A great deal of their life is all about music. When one of them is relaxing in the joint living room of their apartment, it is quite common to find their jukebox playing music. Unfortunately, they do not always share one another's taste in music. Whereas this might be a recipe for endless quarrels in any other situation, there is no arguing over who is in charge of choosing the music being played. This is due to the fact that the jukebox is in fact a small computer (a Mac Mini in our setup) which combines information regarding the presence of its users with their respective musical preference to construct a playlist which is acceptable for all present users. Moreover, if Alice, Jim and Bob invite some friends, their musical preferences can be taken into account as well. Finally, the jukebox also stops playing automatically when it detects that no-one is present. [5]

In this paper we impose an additional constraint on the system. In order to avoid playing the same songs over and over again, the jukebox should avoid playing tracks that its users have heard while they were previously in the room. As we will demonstrate, keeping track of such songs in Crime – such that the reasoning engine can actively use this knowledge – is greatly facilitated by the introduction of Halo's temporal operators. However, before delving into more advanced topics, the next section first presents the core of Crime and Halo.

3 Contextual and History-based Reasoning

3.1 Contextual Reasoning in a Mobile Environment

Crime is a coordination language dedicated to reason about context information in a mobile environment [5]. The language consists of two essential building blocks: a data model and a programming model.

Crime's data model – called the *fact space model* – extends upon the federated tuple space model popularised by Lime [6]. The chief difference between both systems is that tuple spaces correspond to a white board where messages can be published, read and removed, whereas Crime considers a distributed knowledge base describing the context of all nearby devices. The chief *operational* difference between both models is that in a distributed knowledge base both the *assertion* and the *retraction* of a fact are meaningful events which may trigger reactions. The retraction of facts is automatically triggered when context providers disconnect, allowing programs to respond to the (presumed) invalidity of the information they provided.

Crime's programming model consists of a rule-based formalism with a syntax akin to Prolog to describe the causal relation between (a combination of) context events and the corresponding context event handlers. The Crime rules are interpreted by a Rete network which allows for an event-driven and optimised reasoning engine [3]. Rete is a forward-chained algorithm which actively derives every valid conclusion from given a set of facts. As a consequence, Crime applications do not need to manually query the fact space, as the appropriate context event handlers are triggered automatically when the reasoning engine is notified of changes to the fact space.

3.2 History-based Aspects Using Logic

The need to reason about past program state in order to correctly handle events does not only manifest itself in context-aware systems for mobile ad-hoc networks. An interesting parallel can be made with aspect-oriented programming languages offering support for expressing context-dependent behaviour, in e.g. the domain of business rules. One such business rule could be that in an e-shop application, discounts a customer receives upon checkout should depend on whether a discount was active when the user added the item to the shopping cart. This strategy is to be preferred over taking into account discounts active at the checkout, since customers respond badly when they see items they have selected when a discount was active have become more expensive [7, 4].

Similar to event-driven programming approaches like Crime, aspect-oriented pointcut languages allow responding to events (in this case in the program execution) using, for example, a combination of `execution` and `cflow` predicates. However, they typically fall short when events are considered relevant which are no longer active (i.e. they are no longer on the dynamic call stack).

Context-aware aspects introduce an extensible pointcut language where context information can be aggregated into a *context snapshot*. These snapshots can be used to determine whether a pointcut occurs in a conceptual context which is no longer necessarily tied to the dynamic call stack [7]. Whereas that framework is very general, the need to manually snapshot context at certain points in time imposes an imperative style where programmers are actively considering how reasoning about past events should be facilitated.

Halo is an aspect language, built upon the idea of context-aware aspects, yet introducing them in the form of a logic-based pointcut language, enabling a declarative programming style [5]. In Halo, context is modelled as logic facts. Pointcuts can be restricted to such a context, by linking join point conditions to context facts. To make it possible to describe past join points and the past program context in wich they occured, primitives from temporal logic are integrated in the language. Hence pointcuts are aware of the (past) context in which join points occurred

As both Crime and Halo use the Rete algoritm to implement their reasoning engines, it seems plausible that Crime's support for distribution, and Halo's support to manage the fact history can be combined into a single framework.

4 Implementation of the Scenario

We illustrate how the programming model of CRIME accommodates the development of the jukebox application described in section 2. In this section we focus on the actual rules to script the jukebox, and assume the presence of facts of the form `location(‘‘Alice’’, ‘‘DiningRoom’’)` to represent the current location of users and `prefers(‘‘Alice’’, ‘‘Rock’’)` to describe their musical preferences. These facts are published into the distributed knowledge base such that the jukebox application has access to them.

The rule presented below triggers the context event handler `Toggle`. The rule keeps track of the amount of persons which are currently in the jukebox room. When one person is detected the room, this rule will be activated. This implies that the `activate` method of the context event handler will be called, which in turn will start the music player. Similarly, when no-one is left in the room the `deactivate` method will ensure that the music player is stopped.

```
:Toggle() :-
        location(?person, ‘‘Jukebox Room’’).
```

Listing 1.1. `Toggle` Rule

The rating the jukebox attributes to a particular genre depends on both the current number of people in the room and the number of people who prefer the genre. The following two rules calculate these two values by making use of the findall and bagof constructs borrowed from PROLOG. The `findall` construct used in the `total` rule, accumulates all persons located in the room in the *?persons* variable. Similar to the `findall`, the `bagof` construct used in the `category` rule also accumulates all persons in the room but groups them according to the specific genre they like.

```
total(?quantity) :-
        findall(?person, (
                location(?person, ‘‘Jukebox Room’’)),
                ?persons),
        length(?persons, ?quantity).

category(?genre, ?quantity) :-
        bagof(?person, (
                location(?person, ‘‘Jukebox Room’’),
                prefers(?person, ?genre)),
                ?persons),
        length(?persons, ?quantity).
```

Listing 1.2. `total` and `category` Rule

The quantities calculated by both rules presented above are used to trigger the `UpdateRatings` context event handler provided by the jukebox. This event handler will update the ratings of the songs according to the present users' combined preference. These ratings are used in turn by the music player to compile a playlist where highly rated music is featured more often.

```
:UpdateRating(?genre, ?rating) :-
        category(?genre, ?absolute),
        total(?total),
        rating is ?absolute / ?total.
```

Listing 1.3. `UpdateRatings` Rule

4.1 Introducing HALO's Temporal Operators in CRIME

To the best of our knowledge, contemporary frameworks for the development of distributed context-aware applications do not provide reified support to reason about past contexts. In contrast with Halo, reasoning about the past is done manually by recording and manipulating past events in the code of the context event handlers. The current incarnation of the Crime coordination language, as described in section 3.1, exhibits the same shortcoming. However, its event-driven reasoning engine makes it a suitable candidate to introduce the temporal operators developed in Halo.

As a starting point, we propose to introduce the following set of temporal operators from Halo. Note that temporal operators are always implicitly parametrised by the fact that precedes them. That is to say, they have implicit access to the timestamp t_1 at which this fact was triggered.

sometime-past This operator takes one explicit argument (timestamped with t_2) and allows only matching facts such that $t_1 > t_2$. In other words, a rule body of the form `f1(), sometime-past f2()` only matches facts `f2` which occurred **before** a matching fact `f1`.

most-recent This operator has similar semantics as `sometime-past` with the explicit restriction that only one matching fact can be returned. In other words, a rule body of the form `f1(), most-recent f2()` only matches **a single** fact `f2` which occurred **before** a matching fact `f1`.

since This operator takes two explicit arguments (respectively timestamped with t_2 and t_3) and matches facts such that $t_1 > t_3 > t_2$. In other words, a rule body of the form `f1(), since (f2(), f3())`, matches events `f3` which occurred **between** `f2` and `f1`.

Fig. 1. HALO's temporal operators.

These three operators provide an expressive set of building blocks to identify relevant past events. To illustrate this, we complete the scenario described in section 2 by automatically removing songs from the playlist which a user has heard when he has last seen in the jukebox room. The code excerpt below is an outline for a possible implementation.

```
1  :DeleteFromPlaylist(?person, ?songs) :-
2          location(?person, ``Jukebox Room''),
3          most-recent( not location(?person, ``Jukebox Room'') ),
4          since (
5                  most-recent( location(?person, ``Jukebox Room'') ),
6                  findall( ?song,
7                          played(?song),
8                          ?songs).
```

Listing 1.4. Implementation using temporal operators

The rule in the code excerpt is triggered whenever a person enters the jukebox room (line 2). At this point in time, the system recalls the last time when the person left the jukebox room (line 3). This timestamp is used as the end of a `since` interval (line 4), the starting point of which is the previous time the user was spotted by the system (line 5). The fact being sought for in this interval is a `findall` which accumulates all songs played in the interval (lines 6-8). These songs are then deleted from the current playlist (using the `DeleteFromPlaylist` context event handler) as they should not be repeated (line 1).

5 Position Statement

This position paper has identified the need for mobile context-aware applications to be able to reason about past events in order to better adapt their behaviour to the current context. Rather than deferring the reasoning to explicit checks in the context event handlers, we advocate the use of a logic coordination language which incorporates temporal operators as basic language constructs. Such temporal operators have already proven their merit for aspect-oriented programming, a setting which is not dissimilar from the one proposed in this paper. We therefore consider them to be a prime candidate for inclusion in context-aware application toolkits and intend to prepare a proof-of-concept implementation which combines features of Crime and Halo to be presented at the workshop. With this experiment, we intend to contribute a discussion of problems related to integrating temporal reasoning in mobile ad-hoc networks (e.g. volatility and management of distributed historical data etc.).

References

1. J. E. Bardram. *The Java Context Awareness Framework (JCAF) A Service Infrastructure and Programming Framework for Context-Aware Applications.* 2005.
2. P. David and T. Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceeding of MPAC'05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, 2005.
3. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In J. Mylopoulos and M. L. Brodie, editors, *Artificial Intelligence & Databases*, pages 547–557. Kaufmann Publishers, INC., San Mateo, CA, 1989.
4. C. Herzeel, K. Gybels, P. Costanza, and T. D'Hondt. Modularizing crosscuts in an e-commerce application in lisp using halo. ILC 2007, 2007.
5. S. Mostinckx, C. Scholliers, E. Philips, C. Herzeel, and W. D. Meuter. Fact spaces: Coordination in the face of disconnection. In *Proc. of 9th Int. Conf. on Coordination Models and Languages*, 2007.
6. G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *International Conference on Software Engineering*, 1999.
7. É. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. Proc. of the 5th International Software Composition Symposium, 2006.