Declarative Definition of Contexts with Polymorphic Events

Angel Núñez and Jacques Noyé
École des Mines de Nantes
ASCOLA Research Group (EMN-INRIA, LINA)
{angel.nunez, jacques.noye}@emn.fr

Vaidas Gasiūnas
Technische Universität Darmstadt
gasiunas@informatik.tu-darmstadt.de

ABSTRACT

This paper introduces a new model of event handling combining explicitly triggered events with events intercepted with aspect-oriented features. The model supports event abstraction, polymorphic references to events, and declarative definition of events as expressions involving references to events from other objects. We show that this model makes it easy to define a declarative and compositional notion of *event-based context*. We illustrate these ideas with examples in ECaesarJ, a language with concrete support for our model, and relate the events of ECaesarJ to other event-handling and context-handling models.

1. INTRODUCTION

A *context-aware application* is an application that is able to adapt its behavior in order to best meet its users' need. It does so by taking into account *context* information, *i.e.*, any piece of information *relevant to the interaction between a user and an application* [3]. This typically includes information on the physical environment (*e.g.*, noise level, time of day, location, computer resources) as well as the social environment of the user (*e.g.*, nearby people, previous interactions, objectives, mood).

The behavior of context-aware applications depends on the environment context. Programming such applications in a modular way requires modularization of the global context into more specific contexts and attaching specific behavior to these contexts. In other words, a notion of context has to be modular and composable.

Some applications adapt their behavior by determining the current context to decide whether context-dependent functionality should be executed. Other context-aware applications are *reactive* and trigger specific functionality in case of context change.

When dealing with reactive context-aware applications, it makes sense to describe (specific) contexts in terms of changes to the overall context information (the global context). Moreover, specifying these changes as *events* introduces a notion of *event-based context* that links context-aware programming with event-based programming and makes it straightforward to define contexts that depend on events observed in an application.

A modular notion of context is still difficult to provide when the context depends on events happening at different places in the application. This paper introduces a new model of event-handling, combining aspect-oriented techniques with references to explicit events, which makes it easy to define event-based contexts declaratively and compositionally, and thus enables modular programming of reactive context-aware applications.

Section 2 further elaborates on the need for modular, event-based contexts. Section 3 presents the basic features of events as available in ECaesarJ, a concrete implementation of the proposed ideas as an extension to CaesarJ, and explains how they support definition and usage of event-based contexts. Section 4 discusses more advanced issues, showing in particular how events can be defined in a declarative way in terms of other events, thereby supporting composable contexts. Section 5 relates our approach to other event-handling and context-handling approaches. Section 6 concludes.

2. MOTIVATION

A typical example of a context-aware application is software for building automation. Such an application is responsible for smart control of various devices within the building: windows, lights, heaters, hi-fi equipment in a private house, and so on. The behavior attached to the devices depends on specific conditions within the building, which we identify as different contexts. For example, sound notifications make sense only when someone is at home; artificial lighting is necessary when it is dark. So we can identify different contexts, such as "someone at home" and "darkness", which significantly influence the behavior of the system.

A behavior may depend on a context in two different ways.

First, the reaction to certain events or behavior of various processes may depend on the currently *active contexts*. For example, when a motion is detected in front of a house, the front light may be turned on only if it is dark, i.e. if the context "darkness" is active. Similarly, an important notification, of a postman for instance, may be announced by voice if the context "someone at home" is active, while otherwise it would be sent to the cell-phone(s) associated to the house.

Second, it may be important to react to *context changes*. For example, lights must be turned on when it gets dark, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
COP '09, July 7, 2009, Genova, Italy
Copyright 2009 ACM 978-1-60558-538-3/09/07 ...$10.00.

turned off when it gets light again. In such situations it is necessary to react to the changes of the context "darkness", i.e. the lights must be turned on when this context gets active and turned off when it is deactivated.

2.1 Events Provide Context

Dependency on active contexts can be implemented by conditional statements that check which contexts are active and depending on this execute one behavior or the other. The difficulty lies in implementing a condition to check if a particular context is active. The computation using the current program state to determine whether the context is active may be complicated and inefficient. More importantly, the *current* program state may be insufficient to determine whether a context is active, and historical information about events may be required. For example, if it was not recorded that someone entered a house (a door was opened), it may be impossible to determine this fact afterwards (assuming there is no movement sensor). In such cases the application must explicitly keep track of the active contexts by observing the relevant events.

Moreover, the solution based on conditional statements is not suitable for expressing reactions to context changes. In such situations we are interested not in the value of the condition specifying when a context is active, but in the changes to this value. Thus, for each context, we are interested in events indicating when the context is activated or deactivated. Actually, these events define the context.

As we can see, in a lot of cases we need events to express behavior of context-aware applications. The problem is that conventional programming languages lack good support for defining events and the corresponding reactions. For example, in Java applications, events are mostly encoded by the Observer design pattern [4]: The available events are declared as methods of various observer interfaces; the events are triggered by calling these methods, and reactions to the events are defined by implementing the methods.

One problem with the implementation of events by design patterns is that such an implementation introduces a considerable overhead on the required amount of code, related with registration and notification of observers. This problem can be alleviated by dedicated language support for these mechanisms, e.g. events and delegates in C#, but another and more important problem is that a declarative definition of events is not supported. The occurrences of an event are defined by explicit notification calls, i.e. by explicitly triggering the event, which is problematic in two respects.

First, the events defined in this way are difficult to reuse for defining other events. Indeed, it is natural to define events as conjunctions or disjunctions of other events, or by constraining some other events with additional conditions. For example, an event denoting that someone enters a house could be defined as a disjunction of the respective events of multiple outside door sensors. Furthermore, the event "someone enters a house" can be constrained with various conditions to define further useful events, e.g. if nobody is at home this event would mark the beginning of the context "someone at home", or if the alarm system of the house is armed the event would be interpreted as an "intrusion" event and denote the beginning of the corresponding context.

Second, since events are defined by notifications they are defined from the perspective of the notifier. In many cases it is more convenient to define an event from the perspective of the observer, because then we can use the attributes and relationships of the observer for the definition of the event. For example, to define the context asserting that a given person is at a given location, an object implementing this context needs to observe the event "*the* person enters *the* location". A location can notify only about the event when it is entered by *any* person. The object defining the context would have to specialize such an event for the given location and the given person.

2.2 Context Modularization

As described in [10], the definition of when to apply a context-dependent behavior and the definition of a context should be separated so that the definition of a context and its attached behavior can be independently reused and evolved.

We would like to promote the use of an abstract notion of context in order to improve the level of decoupling between the context definition and its use. The first benefit of such a decoupling is the possibility to reuse a behavior with several contexts. Indeed, the use of an abstract context makes it possible to postpone the decision on the concrete context to be used, providing a more stable design. The second benefit is the possibility to reuse the same context with several behaviors, which are not hardwired to the use of specific contexts. We will see that abstract contexts also make it possible to define generic compositions of contexts.

In an object-oriented language, such a decoupling can be achieved by using polymorphism. An abstract class `Context` can be defined for representing a generic context. Context-dependent applications can use objects of type `Context` representing different contexts. A method `isActive` defined in the class `Context` can be used to determine if contexts are active in order to trigger the proper behavior.

In order to attach reactive behaviors to context changes, in addition to the method `isActive`, a context has to declare the events indicating context changes in its interface. Thus, for abstracting from concrete contexts we need a means to abstract from concrete events. In order for a reactive behavior to be reusable with different contexts it must be defined in terms of such abstract events. Furthermore, in order to maintain composability of contexts, these events must also be composable.

3. SIMPLE CONTEXTS

We have refined and implemented the previous ideas in an extension of the programming language CaesarJ [1]: ECaesarJ. ECaesarJ combines the virtual classes and propagating mixin composition of CaesarJ with polymorphic events and support for hierarchical state machines. This section describes how simple contexts can be defined and used based on ECaesarJ events.

3.1 Explicit Events

In a general sense an *event* denotes something of interest happening in time. An event has a *source* (or a *subject*) and *destinations* (or *observers*) interested in the event. From the point of view of the source, events can be *explicit* or *implicit*. Explicit events are explicitly triggered by the source and serve as notifications to the destinations that need to react to a change in the source. Besides explicitly triggered events, we can consider all identifiable changes of program state as implicit events. Examples of such events

are method calls or value changes of object attributes. In aspect-oriented languages, the dynamic occurrences of such events are known as *join points*.

In ECaesarJ explicit events are triggered using a syntax similar to that of method calls. A class (source of events) can trigger events, whose signature (optional return type, name, and formal parameters) is declared in the class. For example, the class below uses the class `java.util.Timer` to generate an event `time` each day at a fixed time.[1]

```
cclass TimeService extends TimerTask {
  static long DAY = 1000 * 60 * 60 * 24; // day in ms
  event time();
  TimeService(Calendar time) {
    new Timer().schedule(this, time.getTime(), DAY);
  }
  public void run() { time(); }
}
```

A class interested in the event, triggered by a specific object, can then define a reaction to it through the definition of an *event handler* associated to the event. The syntax of an event handler is as follows: *expr*`.`*event* `=>` *block*, where *expr* is an expression returning the object that defines the event, *event* is the event, *block*, the reaction to the event. For instance, a class `Rooster` can play a song every morning when it is time to wake up as shown below, where `sts` is set up with the proper time service.

```
public cclass Rooster {
  TimeService sts;

  Rooster(SunTimeService sts) {
    this.sts = sts;
  }
  sts.time() => {
    // play the Rooster Song
  }
}
```

Note that events provide a form of *implicit invocation*: unlike a method caller, which knows the callee, the event source does not know the destination of the event. On the opposite, the event destination knows about its source.

3.2 Context Definition

In ECaesarJ, an object can use events defined by other objects. We use this feature to implement contexts in a modular way, as objects defining pairs of events `in` and `out` that activate and deactivate the contexts, respectively. In addition, a context provides a method `isActive`, which makes it possible to find out whether the context is currently active.

Since, as we will soon see, events can be defined not only by explicit triggering, but also in other ways, an event can be declared as `abstract` to abstract from the way it is implemented in subclasses. In a class `Context`, describing an interface common to all specific context definitions and giving its partial implementation, we declare events `in` and `out` as abstract, so that they can be defined differently in the subclasses of `Context`. The abstract class `Context` is given below.

```
1  abstract cclass Context {
2    abstract event in();
3    abstract event out();
4
5    boolean active;
6
7    boolean isActive() { return active; }
8
9    in()  => { active = true; }
10   out() => { active = false; }
11   ...
12 }
```

A context maintains a boolean variable indicating the state of the context: active or inactive. The language constructs of lines 9 and 10 define event handlers that update the state of the context when one of the events `in` and `out` occurs. Since the state of the context is tracked, the implementation of the method `isActive` is obvious, avoiding evaluation of (potentially expensive) conditions to determine whether the context is active.

A concrete context must provide a concrete definition for the events `in` and `out`. For example, the class `ScheduledNightTime` below implements a concrete context by defining the events `in` and `out` as the events `time` triggered by the time services `sunset` and `sunrise`.

```
cclass ScheduledNightTime extends Context {
  TimeService sunset, sunrise;

  event in() = sunset.time();
  event out() = sunrise.time();
}
```

3.3 Context Use

A context-dependent class can maintain one or more references to context objects, and express context-dependent behavior as reactions to these events or by using the method `isActive`. A context-dependent class can also define its own events in terms of the events defined (provided) by a context. This provides a form of aliasing. For example, the listing below shows the implementation of a class `LightAutomation`, which defines the events `mustTurnOn` and `mustTurnOff`, triggered when the light must be turned on and turned off, respectively. The class is parameterized with an abstract context, so that these events can be defined as the activation and deactivation of the context.

```
1  abstract cclass ILight {
2    abstract void switchOn();
3    abstract void switchOff();
4  }
5
6  cclass LightAutomation {
7    ILight light;
8    Context context;
9
10   event mustTurnOn() = context.in();
11   event mustTurnOff() = context.out();
12
13   mustTurnOn()  => { light.switchOn(); }
14   mustTurnOff() => { light.switchOff(); }
15   ...
16 }
```

The previously defined context `ScheduledNightTime` can be used together with instances of the class `LightAutomation` of the previous section in order to turn off some lights when it is dark.

```
LightAutomation automator = new LightAutomation();
automator.setLight(...);
```

[1] We use the keyword `cclass` for classes supporting specific ECaesarJ features. The keyword `class` is reserved for pure Java classes for backwards compatibility. In the listings we skip visibility annotations for the sake of simplicity.

```
automator.setContext(new ScheduledNightTime());
```

The use of a class parameterized with an abstract context makes it possible to instantiate the class `LightAutomation` with different contexts. For example, an instance can be created for a light in the living room when it is dark. Another instance can be created for the front light in case there is a motion detected in front of the house at night.

Note that such reuse is enabled by the fact that references to events are polymorphic, i.e. they can be bound to different concrete event definitions, depending on the dynamic type of their owner object. For example, in line 10 of the above example, the event `context.in()` is used without knowing its concrete definition: it definition depends on the dynamic type of the variable `context`.

4. ADVANCED CONTEXT DEFINITION

We have seen so far that events could be explicitly triggered and aliased. But they can also be refined using conditions as well as implicitly triggered. This section details these advanced features, including quantification over implicit events and quantification over a list of events.

4.1 Event Refinements

Within an event destination, the event can be refined by attaching conditions on parameters exposed by the event and on values computed in the scope of the enclosing class. In addition, an event can be defined as a composition of other events. Note that, on the destination side, an explicit event can actually be seen itself as a condition: the fact that the event is indeed received by the destination.

As an example, the listing below shows a class `SensorNightTime`, an alternative implementation of a context representing night time, using a light sensor to measure daylight intensity. The context observes the event `intensityChanged` on the sensor in order to determine when it is getting dark. The event `in` is defined as the occurrence of the event `intensityChanged` when the context is inactive and the light intensity is lower than a given threshold. The deactivation event is defined in an analogous way.

```
cclass SensorNightTime extends Context {
  LightSensor sensor;
  int threshold;

  event in() = sensor.intensityChanged(int i) &&
    if(!isActive() && i < threshold);

  event out() = sensor.intensityChanged(int i) &&
    if(isActive() && i > threshold);
  ...
}
```

4.2 Implicit Events

In addition to explicitly triggered events, we also consider identifiable points of execution as implicit event occurrences. An implicit event is defined using AspectJ-like pointcuts. As an example, the listing below illustrates a third alternative for a context denoting night time. In this case, night time can be detected indirectly considering that blinds are automatically closed at night time and opened at day time. Thus, the events `in` and `out` are defined as the execution of the methods that close and open blinds, respectively.

```
cclass BlindNightTime extends Context {
  event in() = execution(* *.closeBinds(..));
  event out() = execution(* *.openBinds(..));
}
```

4.3 List Quantification

Since implicit events include AspectJ-like pointcut designators, this kind of event definition supports quantification over program structure. In addition to this form of quantification, ECaesarJ supports quantification on a list of objects. An event can be defined as a disjunction of events with the same name. This is done using an expression of the form **some**(*list*) . *evt*(*params*), where *list* denotes a list of objects, and *evt* an event defined by these objects.

This form of quantification is necessary for defining contexts that depend on the state of multiple objects. When computating the condition checking whether a context is active, it is possible to iterate over multiple objects and query their state. List quantification gives an analogous possibility for event-based definitions of contexts: the events denoting the beginning and the end of a context can be defined in terms of events of multiple objects. For example, the context of the presence of somebody in a location can be described in terms of events of motion sensors available in this location. A motion sensor provides an event `motion`. The constructor **some** is very useful in this case. As illustrated in the example below, the presence context is activated when **some** `motion` is detected:[2]

```
cclass PresenceAtLocation extends Context {
  Location location;
  TimeService timeOut;
  event in() = some(location.motionSensors()).motion();
  event out() = timeOut.time();
  ...
}
```

4.4 Context Composition

As described in [10], contexts need to be composable.

For example, we may need a context, in which somebody is at a location and it is dark, in order to turn on the lights at the location. This implies a context that is the conjunction of the contexts `SensorNightTime` and `PresenceAtLocation` of the previous section.

Contexts can be composed by combining the activation and deactivation events of these contexts. Thanks to polymorphic events, different kinds of context compositions can be implemented in a generic way. The example below describes the conjunction of two contexts as a new context `AndContext`.

```
cclass AndContext extends Context {
  Context ctx1, ctx2;
  event in()  = ctx1.in() && if(ctx2.isActive()) ||
                ctx2.in() && if(ctx1.isActive()) ||
                ctx1.in() && ctx2.in();

  event out() = ctx1.out() && if(ctx2.isActive()) ||
                ctx2.out() && if(ctx1.isActive());

  AndContext(Context ctx1, Context ctx2) {
    this.ctx1 = ctx1; this.ctx2 = ctx2;
  }
}
```

[2]The event `out` is defined as occurring a fixed amount of time after the last detected motion. For this we use the class `TimeService` presented in the previous section.

In a similar way other generic context compositions can be defined, such as a disjunction, negation, and difference of contexts.

5. RELATED WORK

Context-Handling Approaches

In [10], Tanter *et al.* advocate separating the definition of a context from its use so that attached behavior can be independently reused and evolved. They use their notion of context to introduce *context-aware aspects* as aspects that match base-program join points depending on whether a given context is active. As in *context-aware aspects*, our notion of context is explicit, composable, and can be activated by observing join points. However, we explicitly introduce events, which includes not only join points but also explicitly triggered events. In addition, we modularize the definition of the events that activate and deactivate a context.

ContextL [2] is an Object-Oriented Programming language targeted at Context-Oriented Programming [6]. It provides means to associate partial classes and method definitions with *layers* and to activate and deactivate such layers in the control flow of a running program. Thus, the behavior of objects is extended with the activated layers. Our approach is not comparable with ContextL as we deal with different issues. ContextL deals with context-dependent activation layers of program structure and behavior, while our focus is more on declarative definition of the contexts themselves. In ContextL, the active contexts correspond to active layers. Since layers are activated following a stack discipline, (de)activation of contexts is more restricted than the (de)activation of the contexts presented in our approach.

In previous work [8], we presented a model for event-based contexts, in which we proposed to define contexts in terms of context activation and deactivation events. The focus was both on context modularization and on modeling the reaction to context changes using a process algebra, making it possible to detect property violations. This paper makes the notion of context presented in [8] concrete by using the events of ECaesarJ.

Event-Handling Approaches

Events identify phenomena happening in an application and the environment, which can determine different contexts. Usually, some source components trigger events and destination components handle them. For modularization reasons, event sources are expected to be as much as possible unaware from the destinations of these events. The observer design pattern [4] makes this possible by injecting, within each potential source of events, an infrastructure providing a registration service for the interested observers as well as a notification service for the source, which, as a result, does not need to explicitly invoke the observers, destinations of the event. This realizes a weak form of Implicit Invocation (II). Implicit Invocation was initially introduced in [5] as an architectural style in which event sources are completely unaware of event destinations with registration and notification supported behind the hood. Our proposal uses a form of implicit invocation. However, in our case, an event triggered by a source can be seen as several events from the point of view of the destinations. These views can be composed in a declarative way and can be reused.

In Aspect-Oriented Programming (AOP) [7], join points can be seen as events that are implicitly triggered in the execution of an application, and advices can be seen as handlers of such events. Although join points are again defined from the point of view of the destination, there is no explicit notion of event that can also be explicitly triggered. Unlike pointcuts, the events in ECaesarJ are late-bound, which enables reuse of event of other objects, as well as abstraction from these events.

Ptolemy [9] is a hybrid approach between II and AOP. Events of Ptolemy are associated with code-snippets, which can be seen as explicit triggering of join points. Classes can define bindings to such events or to compositions of multiple events. Ptolemy, however, does not provide features for event abstraction and declarative definition of events: the events are declared globally and can be defined by explicit triggering only.

Like in ECaesarJ, C# events are also declared as members of objects and can be explicitly triggered. C# however does not support declarative definition of events in terms of other events and join point interception. Events of other objects can be used only by explicitly registering to them.

6. CONCLUSION

We have shown how ECaesarJ events support declarative and modular definition of event-based contexts. Direct support for events in a programming language, supplied with dedicated referencing and composition mechanisms, facilitates natural and concise definition of contexts in terms of events. Polymorphic events enable decoupling of context definition from its use, thereby supporting independent evolution and reuse of both the context definitions and the behaviors attached to the contexts. Finally, the possibility of composing polymorphic references to events can be exploited for generic composition of contexts.

In this paper, a conjunction of events is limited to a conjunction of conditions applying to the same program execution point. ECaesarJ makes it also possible, using state machines, to define the conjunction of events triggered from different program execution points, which is also useful for defining the events that activate and deactivate contexts, as shown in [8].

A number of ECaesarJ constructs, including events, event handlers, and state machines have been inspired by construct and concepts used in concurrent languages. The initial model, developed in [8], was also inherently concurrent. A natural step would therefore be to enhance ECaesarJ with concurrent facilities.

Acknowledgments

This work has been partly supported by the European project AMPLE: Aspect- Oriented, Model-Driven, Product Line Engineering (STREP IST-033710).

7. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, Feb. 2006.

[2] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of

ContextL. In *DLS '05: Proceedings of the 2005 Symposium on Dynamic Languages*, pages 1–10. ACM Press, 2005.

[3] A. K. Dey and G. D. Abowd. Towards a better understanding of context and contex-awareness. In *Proceedings of the CHI 2000 Workshop on the What, Who, Where, When and How of Context-Awareness*, The Hague, The Netherlands, Apr. 2000. Georgia Tech.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, volume 551 of *lncs*, pages 31–44, Noordwijkerhout, The Netherlands, 1991.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming - 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[8] A. Núñez and J. Noyé. An event-based coordination model for context-aware applications. In D. Lea and G. Zavattaro, editors, *10th International Conference on Coordination Models and Languages (COORDINATION 2008)*, volume 5052 of *Lecture Notes in Computer Science*, pages 232–248, Oslo, Norway, June 2008. Springer-Verlag.

[9] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008)*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179, Paphos, Cyprus, July 2008. Springer-Verlag.

[10] E. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. In W. Löwe and M. Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 227–242, Vienna, Austria, Mar. 2006. Springer-Verlag.