Addressing Dynamic Contextual Adaptation with a Domain-Specific Language

Serena Fritsch, Aline Senart, Siobhán Clarke
Distributed Systems Group, Trinity College Dublin, Ireland
fritschs, senarta, sclarke@cs.tcd.ie

Abstract

The increasing number of mobile devices and sensors equipped with wireless networking capabilities enable a new generation of pro-active applications. These applications make use of context to adapt their behaviour to better fit their current situation. To support unanticipated changes to application behaviour, mechanisms are needed to specify when and how to adapt an application during its runtime. Many dynamic platforms exist that achieve this to some extent, and that are built on general-purpose languages (GPLs). However, these approaches suffer from standard difficulties of GPLs relating to the lack of semantic expressiveness of their constructs. In this paper, we describe high-level declarative constructs that can be used to specify the adaptation of application behaviour to specific situations. The language is supported by a framework that enables the exchange and merge of behaviours on-the-fly. Our approach is evaluated against application scenarios in the domain of autonomous vehicles.[1]

1 Introduction

In a near future, a multitude of sensors will be deployed in the physical world to gather information about the changing environment in which applications execute and various communication technologies will be available to support rich interaction between these applications across space and time. This demands a change of paradigm in application development towards self-adaptable and autonomous applications that operate in a highly mobile environment. These applications particularly need to adapt their behaviour to adjust to the current situation [5].

Writing adaptive applications remains an arduous task for application developers. Existing approaches such as the strategy pattern and the decorator pattern [10] demand the design of all adaptational parts of an application at design time. However, it is not feasible to anticipate all the situations in which an application may be required to adapt. Hence, mechanisms are needed that allow the dynamic change of application behaviour at runtime.

Existing dynamic adaptation mechanisms, such as dynamic aspect-oriented programming [13], provide means for application developers to state when and how behaviour of an application should be adapted. However, these approaches usually rely on general-purpose language constructs and do not allow programmers to reason at a higher semantic level. Hence, application developers have to express adaptations in the format of the supporting platform which often remains difficult.

Domain-specific languages [12] offer a solution to raise the semantic level of expressivity. These languages are tailored to a specific application domain with expressive mechanisms for application developers that result in a higher semantic level of reasoning about concerns of a domain. In our approach, we propose a language that provides higher-level constructs for expressing the adaptation of application behaviour due to a change in context. Application behaviour is modelled with state machines. State machines are a well-known technique to describe the behaviour of applications and provide abstractions that enable code maintenance and reuse [8]. By applying this technique to pro-active applications, we define a context graph where a state describes contextual behaviour of an application and transitions between behaviours are triggered depending on new situations. Using context graphs, contextual behaviours can be separated and better understood, while allowing their reuse and future extensions.

The rest of this paper is structured as follows: Section 2 discusses related work. Section 3 motivates the needs for a declarative language with two scenarios. Section 4 describes our approach in detail. Section 5 concludes this

paper.

2 Related Work

Adaptation is a challenging requirement in mobile and pervasive computing. This section discusses several approaches that provide programming abstractions for dealing with the adaptation of an application to changing situations. Most of these approaches enable applications to adapt to changing system resources/services by selecting dynamically the resources/services to use based on their quality and availability. In contrast to them, our language focus on the actual change of the application logic during runtime. In the following discussion, we distinguish between component-based ([2], [15]), middleware ([4], [6]), linguistic ([11], [1], [9]) and technique-based approaches ([14]).

Component-based approaches, like PCOM [2], follow a generic automatic adaptation approach, i.e., the application developer specifies functional and non-functional properties of services that are required by an application and defines additional adaptation goals. An adaptation goal contains the adaptation preferences of the application, e.g., choose dynamically the service with the highest available resolution. Adaptation is applied when a change in one of the required services occurs. The system then tries to automatically find the most suitable service at runtime.

Gaia [15] provides programming abstractions for developers to define the required functionality of an application on an abstract level. Developers can specify active space entities and common operations on these spaces. An active space is a physically bounded collection, such as a room of devices, applications and services. During runtime, these entity definitions are bound to actual available services. Adaptation occurs when the user moves from an active space to another. Gaia and PCOM address the adaptation of the services available to the application rather than the adaptation of the application logic itself.

The authors of [4] propose a reflective middleware architecture that uses reflection and context-awareness to support the adaptation to a changing context. Adaptation decisions are encapsulated inside XML-based user profiles and the appropriate reconfigurations are redirected to the basic application code. However, the reconfiguration actions address changes to services, e.g., communication and their delegation to user code through callbacks makes them hard to understand.

Ledoux et al. [6] present a framework for self-adaptive components. Adaptation logic is separated from the business parts of an application. The specification of the adaptation logic follows the Event-Condition-Action pattern and makes use of events emitted by a context-aware service that provides information about the execution context of the application. Their approach is rather focused on the adaptation of system services, e.g. the size of the cache, than the change of application behaviour.

The Container Virtual Machine (CVM) is a generic adaptation language for component-based applications [11]. The language provides domain-specific constructs for describing the adaptation of system services of components, e.g., adding an encryption service. Each adaptation can be translated to different target platforms, depending on the current execution platform of the application. However, the language does not explicitly support the change of application logic.

Babu et al. [1] provide explicit language constructs for abstracting change of behaviour in objects. Change in behaviour is modelled as first class entity and the basic application is annotated with specific constructs to weave the additional behaviour during runtime. The base code can then become cluttered with many annotations, interfering with its readability.

Fuentes et al. [9] describe an XML-based domain-specific language that allows to write adaptation strategies. The approach is based on the DAOPAmI platform, a component- and aspect-based platform that provides a list of common services for ambient intelligence applications. However, the language only considers the architecture of an application and the adaptation of system services, like cache size and communication mechanisms, to better fit a current situation.

The authors of [14] propose a new approach based on technique-based programming, in which goals represent abstract services and techniques are selected at runtime to fulfil these goals. A runtime process supports the adaptation of the application to changing availability in resources. This approach focusses on the adaptation of an application to changing resources in a mobile environment.

3 Application Scenarios

In this section, we study two scenarios from the domain of autonomous sentient vehicles in order to derive the requirements that our language should fulfill. Both scenarios require applications to adapt their behaviour during runtime of the application. In order to describe these scenarios in detail, we will first define a number of terms.

We define behaviour as "a set of actions and reactions of an application in a specific situation". A change in a situation requires a change in the behaviour of an application and hence in the actions to take.

The various behaviours of a context-aware application can be modelled as a context graph [16]. A context graph represents a finite state automaton and encapsulates application behaviour into a set of distinct context states. Therefore, it provides a useful and modular abstraction to structure complex, context-aware applications [3].

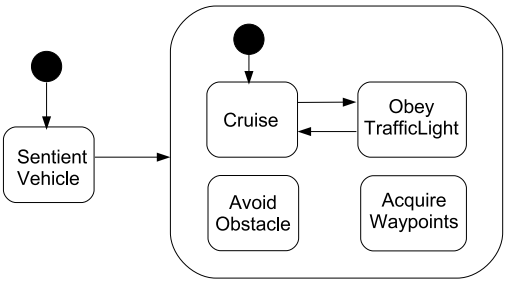

Figure 1. Context graph for an autonomous vehicle

A context state in a context graph corresponds to a specific situation that is determined based on input from sensors. Each state contains a set of appropriate actions that characterise the contextual behaviour to be carried out by an application, and a set of context variables that are valid in this state. The values of context variables can be either static, e.g., the name of an entity, or can change continually, e.g., the position of a mobile entity.

The presence of a new situation requires the alteration of the present course of actions between context states. These transitions are based on the evaluation of transition rules that assess the values of state variables.

Figure 1 depicts an example of a context graph for an autonomous vehicle with the states *Cruise*, *AcquireWaypoint*, *AvoidObstacle* and *ObeyTrafficLight*. This context graph determines the default behaviour of a vehicle, e.g., in the *Cruise* state, the vehicle travels in a straight line, accelerating to cruising speed. When the vehicle is in the state *AvoidObstacles* after it has detected an obstacle on the road with its sensors, it either brakes or changes direction. Transitions between states are triggered by events, e.g., the detection of a traffic light leads to a transition from the *Cruise* state to the *ObeyTrafficLight* state. Note that for reasons of clarity, we omit other transitions in Figure 1.

3.1 Exchanging Behaviour

Figure 2 presents a scenario where there is a need for an autonomous vehicle to change its default behaviour based on an unplanned situation. After the arrival of an emergency vehicle, the vehicle has to adapt its behaviour to avoid collisions with the ambulance and with other vehicles and free the way. The behaviour for this situation is contained in the context graph *AmbulanceArrival* that contains the context states *GetOutOfTrajectory* and *IsOutOfTrajectory*. The states contain actions specific to this new situation.

However, this situation is not reflected in the currently active context graph *SentientVehicle* that describes the default behaviour of a vehicle. In order to react appropriately, an exchange of context graphs has to take place that results

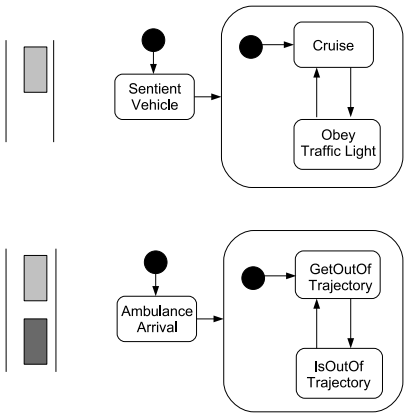

Figure 2. Exchanging application behaviour

in an adaptation of the application to better match the current situation.

In the following, we list a set of issues (Ix) with regards to the above mentioned scenario. We assume that the vehicle is in the *Cruise* state initially. For illustrating purposes, the two context graphs shown in Figure 2 have been simplified.

I1: Point in time of adaptation: A mechanism is needed to specify when an adaptation should take place. In the scenario, an adaptation is necessary when an emergency vehicle arrives.

I2: Applicability of an adaptation: The question arises whether the exchange of behaviour can be executed immediately or need to be delayed until some actions are finished. For example, if the vehicle is sending a message to another entity or is currently braking, the exchange of behaviour has to be delayed until the action has ended.

I3: State conservation: The current active context in the context graph should be saved for later reactivation. As there is a set of variables associated with each state, these variables have to be either saved or migrated to the new behaviour. For example the state variable *destination* can be migrated as it is still valid in the new behaviour. However, variables like *currentSpeed* are not meaningful for the new behaviour and as a result of this, it is not necessary to migrate them.

I4: Determination of the new state: The initial state of the new application behaviour has to be determined. Often the mapping to the state of another context graph is dependent on the values of context variables. For example, in our scenario, whether to map to the initial state *IsOutOfTrajectory* or *GetOutOfTrajectory* depends on the current position and destination of the autonomous vehicle and the emergency vehicle. For this, explicit constructs are needed to specify which variables to consider for the actual exchange of behaviour. Moreover, a mechanism is required that is able to constantly monitor these variables.

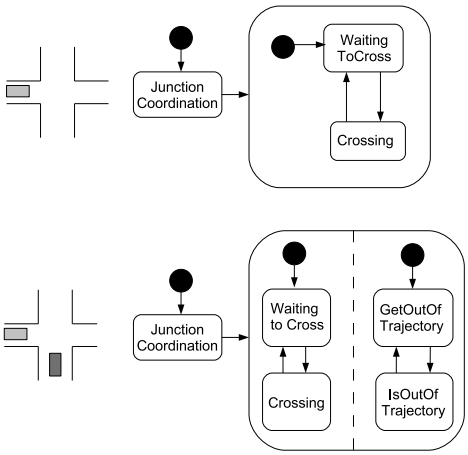

Figure 3. Merging of Behaviour

I5: Duration of adaptation: Some scenarios require that the behaviour of the application switches back to the previous behaviour, e.g., when the emergency vehicle has passed, the SentientVehicle context graph is active again. In these cases, the previous state has to be restored and initialised with consistent values.

3.2 Merging Behaviour

Figure 3 depicts a scenario where two behaviours have to be merged together in order to react to a specific situation. The first context graph shows the behaviour of a vehicle that enters an unsignalised junction and coordinates its behaviour with approaching cars to cross the junction safely. Later, an emergency vehicle arrives raising a need for the vehicle to react accordingly in order not to crash with the ambulance vehicle and to free the way. In this case, a simple exchange of context graphs is not sufficient, as the vehicle still needs to coordinate its behaviour with upcoming vehicles at the junction. This scenario extends the list of issues to the following:

I6: Determination of currently active behaviour: In the scenario, the resulting merged context graph consists of two sub-context graphs, each of which determining the behaviour of a vehicle with regards to other vehicles. For example, when an emergency vehicle arrives, the context graph on the right hand side should be active and the behaviour of a vehicle is determined with regards to this. Hence, a mechanism is needed to determine which sub-context graph to activate based on the type of situation and interacting entity.

I7: Exclusion of states: In some situations, an entity cannot be in two states at the same time as these states exclude each other. For example, the state *Crossing* and the state *IsOutOfTrajectory* are incompatible to each other, as a vehicle cannot at the same time ensure a non-collision with an emergency vehicle while crossing the junction. The question here arises how to formalise these constraints.

I8: State explosion: As there are potentially an unbounded amount of sub context graphs, the merged context graph potentially results in a huge state explosion. A mechanism should be provided that minimises the number of states and sub-context graphs.

3.3 Requirements for the language

From the issues identified above, we derived some requirements (Rx) our language should fulfil:

R1: The language should support the dynamic exchange of context graphs.

R2: The language should support the dynamic merge of context graphs.

R3: The language should provide mechanisms to specify when to exchange or merge context graphs.

R4: The application developer should be able to specify context variables that can be accessed and that can be used at runtime to trigger an adaptation.

R5: The language should provide mechanisms to determine the initial state of the new context graph after an exchange or merge of context graphs.

R6: In the case of a merge, the language should provide mechanisms to express which sub-context graph should be active, dependent on the new situation.

R7: Capturing the state of a context graph is necessary before switching or merging behaviours. Such capture requires to save the value of context variables for each state of the context graph at the time of the capture.

R8: The language should provide constructs for specifying the duration of an adaptation.

R9: The application developer should be able to define constraints such as timeliness or mutual exclusion of states of different context graphs.

R10: The language should support a scalable merging operator for context graphs.

4 Our Approach

In this section, we describe our overall approach to support the above identified requirements (c.f. Figure 4).

The proposed framework consists of the domain-specific language that allows the specification of dynamic changes in application behaviour. Additionally, it defines a mechanism that allows the actual dynamic adaptation of behaviours at runtime and the supervision of the application at runtime. Finally, the context-based programming model is used by application developers to design pro-active applications.

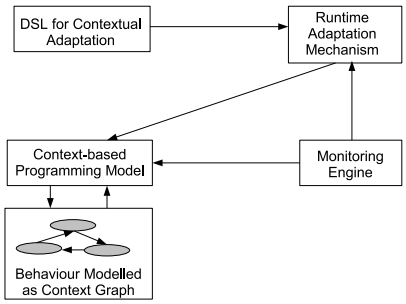

Figure 4. Overall System Architecture

4.1 Domain-specific language for dynamic contextual adaptation

In this subsection, we introduce some of the concepts and notations that are supported by our domain-specific language to express the adaptation of application behaviour at a conceptual higher level. We illustrate some of the concepts by means of the previously discussed scenarios. The mechanism how to define valid configurations and to restrict the possibility of an explosion of context graph states remains an open question and needs further investigation. Subsequently, the following language constructs cover only a subset of the identified requirements.

Listing 1 illustrates the specification for exchanging context graphs, when a message from an emergency vehicle is received. We will refer to parts of this listing when discussing the relevant requirements.

```
1 $entity1 = "sentientCar";
2 $entity2 = "emergencyVehicle";
3 bind $positionCar = $entity1::position;
4 bind $positionEmVehicle = $entity2::position;
5 bind $directionCar = $entity1::direction;
6 bind $directionEmVehicle = $entity2::direction;
7 try until (message == EmergencyVehiclePassed)
8 when (message == EmergencyVehicleArrival) do {
9         exchange behaviour "SentientVehicle" "
              AmbulanceArrival";
10        map to contextState "GetOutOfTrajectory" on
              condition
11        ($positionCar, $positionEmVehicle) < threshold
              &&
12        ($directionCar == $directionEmVehicle)
13        else map to contextState "IsOuOfTrajectory"
14 }
```

Listing 1. Exchange Behaviour Code Example

R1: To support the dynamic exchange of context graphs, we have defined an exchange language construct (see line 9) that takes as input two behaviours, modelled as context graphs, and replaces the first behaviour with the second behaviour. The general syntax for this is as follows:

exchange contextgraph$_1$ contextgraph$_2$

Listing 2 illustrates the specification of the merging of the two context graphs *JunctionCoordination* and *AmbulanceArrival* when an emergency vehicle arrives.

```
1 $entity1 = "sentientCar";
2 $entity2 = "emergencyVehicle"
3 $contextGraph1 = "JunctionCoordination";
4 $contextGraph2 = "AmbulanceArrival";
5 try until (message == EmergencyVehiclePassed)
6 when (message == EmergencyVehicleArrival) do {
7         merge $contextgraph1 $contextgraph2;
8         if $entity1 apply $contextGraph1;
9         if $entity2 apply $contextGraph2;
10 }
```

Listing 2. Merge Behaviour Code Example

R2: To support the dynamic merge of context graphs, we have defined the merging language construct (see - line 7) that takes as input an arbitrary number of context graphs and merges their behaviours into a resulting context graph. The first context graph is replaced with a context graph containing the merged behaviour.

merge contextgraph$_1$ contextgraph$_2$ ...contextgraph$_n$

R3: To enable the application developer to specify when an adaptation has to be performed, we follow the Event-Condition-Action pattern, widely used in active databases [7]. The language construct below denotes a lazy context switch. This construct can be used when the new behaviour should be applied only when the application is in a consistent state and it is safe to execute an adaptation.

try ... when event *do* adaptation

As an example, Listing 1 - line 8 and Listing 2 - line 6 specify that the adaptation should occur when a message from an emergency vehicle is received.

R4: To enable the application developer to specify context variables and to associate them with a specific entity, we have defined the following language construct:

bind variableName = entityName::contextVariable,

where *variableName* is a placeholder of our language, and *entityName* and *contextVariable* refer to an actual context variable of a specific entity.

Lines 3 - 6 in Listing 1 illustrate the use of these constructs, by binding the context variables *position* and *direction* the sentient car and emergency vehicle respectively.

R5: To specify the initial state of the new context graph, we have defined a language construct that expresses what is the initial state and what are the conditions to switch to this state. The proposed general syntax for this is illustrated below:

map to contextState identifier

map to contextState identifier *on condition* conditionIdentifier

The *conditionIdentifier* specifies into which context state

to switch, depending on the values of context variables, e.g., current position and speed of an entity.

For example, lines 9 - 13 of Listing 1 specify the actual exchange of behaviour. The determination of the new state depends on the values of the position and direction variables of the vehicle and ambulance. If the vehicle and the ambulance are heading to the same direction and the distance between them is below a specified threshold, the initial state of the new behaviour will be *GetOutOfAmbulanceTrajectory*, else the initial state will be *IsOutOfAmbulanceTrajectory*.

R6: To enable the application developer to express which context graph to activate in case of a merge of contexts, we provide the following structure that specifies which context graph to apply depending on the interacting entity:

```
if entity apply contextgraph
```

For example, in Listing 2, lines 7 and 8, the active context graph should be either the *JunctionCoordination* or the *AmbulanceArrival* context graph, depending on the entity with which the vehicle interacts.

R7: To capture the state of a context graph, we need to ensure that its state is consistent. Therefore, before a capture, we automatically stop the context graph so that it will not react anymore to incoming events. The events are logged and will be resend when the new context graph is started. We have defined language constructs that allow the application developer to specify stop/start actions as part of an adaptation when the default behaviour is not appropriate. These actions can be applied to either the complete context graph or a specific state.

```
stop contextgraph/state
start contextgraph/state
```

R8: To define until when an adaptation of the application behaviour is valid, we have defined the language constructs illustrated below. These constructs allow the application developer to define until when the adaptation of application behaviour is valid. Validity can either be based on a specific duration or until a specific condition becomes true.

```
try for time when event do adaptation
try until condition when event do
adaptation
```

For example, Listing 1 line 7 and Listing 2 line 5 specify that the adaptation is valid until the emergency vehicle has passed.

4.2 Runtime Adaptation Framework

In this section, we present the framework that supports the domain-specific language and that provides an execution platform for pro-active applications. It is comprised of the Runtime Adaptation Mechanism that dynamically exchanges application code and of the Monitoring Engine that determines when adaptations should occur. Figure 5 illustrates the execution environment of our framework.

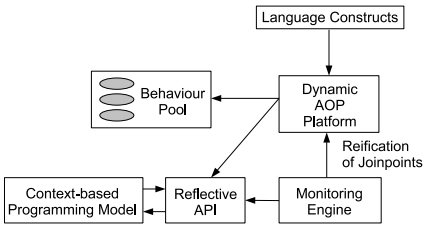

Figure 5. Execution Platform

Runtime Adaptation Mechanism The runtime adaptation mechanism is responsible for adapting the actual behaviour of the running pro-active application. To achieve this, we employ dynamic aspect-oriented programming mechanisms (AOP) [13] that allow the adaptation of an application during its runtime by continuously monitoring its execution and executing pieces of code at specified points in the execution (so-called join points).

In particular, language constructs, like the specification of the adaptation are translated into executable aspects of a general-purpose, dynamic aspect language. An aspect contains the definition of when to apply a specific adaptation of behaviour and the actual exchange or merge of this behaviour. Application behaviours are stored in a behaviour pool. In our approach, adaptations occur when the Monitoring Engine detects a reception of a message, e.g., *emergencyVehicleArrival*. The adaptation aspect then chooses the specified behaviour from the behaviour pool and executes the adaptation.

To execute the actual adaptation, reflective capabilities are needed by the Runtime Adaptation Mechanism. Reflection refers to the ability of a program to introspect and change its own structure at runtime. In our model, the context-based programming model provides reflective capabilities to reason about and change the current behaviour of the application.

Monitoring Engine The monitoring engine monitors the application at runtime and evaluates the conditions on context variables to determine whether to trigger behavioural adaptations. For example, in the above described scenarios, the engine triggers an adaptation that requires a merge or an exchange of behaviour, when an emergency vehicle notifies the vehicle about its arrival by sending a message.

Moreover, the monitoring engine is responsible for monitoring the current values of variables to trigger adaptations. For example, in the first scenario the context variables position and direction of the vehicle and emergency vehicle have to be monitored and their latest values have to be re-

trieved when an adaptation takes place in order to determine what the initial state of the new context graph is.

4.3 Context-based Programming Model

The context-based programming model serves as a building block for pro-active applications. In the model, application behaviour is encapsulated into contexts within a context graph. The granularity of contexts is application-specific. A context can represent a single short action like turning or a more elaborate one like travelling from a source to a destination.

An execution platform will be provided that enables applications designed with the context-based programming model to run. As a starting point, we envisage to use MoCoA [16], a middleware platform, as our execution platform.

MoCoA supports a small set of abstractions that can be used to support a wide range of context-aware (mobile) applications. Applications are structured using the sentient object model [3]. A sentient object is a mobile, intelligent entity that extracts, interprets and uses context information obtained from sensors to derive its behaviour.

Behaviour in the sentient object model follows the context graph paradigm, where each state defines the actions of an entity for a specific situation. The platform offers a well-defined API and is therefore easily expandable.

5 Conclusion and Outlook

In this paper, we have proposed a domain-specific language for describing dynamic contextual adaptations at a higher level. The language constructs support the exchange and merge of behaviours that are specified as context graphs. The requirements for the language constructs were illustrated with two examples in the domain of autonomous vehicles. We have proposed an overall programming framework that consists of the domain-specific language, a runtime adaptation mechanism and a context-based programming model. The runtime adaptation mechanism is responsible for the actual execution of the dynamic adaptation, whereas the context-based programming model defines the underlying building blocks of an application. We are currently investigating how to support constraints on context graphs and how to provide scalable merging of behaviour.

Acknowledgement

This work is funded by Science Foundation Ireland under the Research Frontiers Program.

References

[1] C. Babu, W. Jaques, and D. Janakiram. Dynamic customisation in pervasive computing environments. In *IEEE Conference on Enabling Technologies for Smart Appliances (ETSA)*, 2005.

[2] C. Becker, M. Handte, G. Schiele, and K. Rothermel. Pcom - a component system for pervasive computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, 2004.

[3] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Proceedings of the 2^{nd} IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2004.

[4] L. Capra, W. Emmerich, and C. Mascolo. Reflective middleware solutions for context-aware applications. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflections)*, 2001.

[5] J. Coutaz, J. Crowley, S. Dobson, and D. Garlan. Context is key. *Commun. ACM*, 48(3):49–53, 2005.

[6] P. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. In *Proceedings of Distributed Applications and Interoperable Systems, (DAIS)*, 2003.

[7] K. R. Dittrich, S. Gatziu, and A. Geppert. The Active Database Management System Manifesto: A Rulebase of a ADBMS Features. In *2nd International Workshop on Rules in Database Systems (RIDS)*, 1995.

[8] M. Fowler. *UML Distilled, Third Edition*. Addison-Wesley Professional, 2003.

[9] L. Fuentes and D. Jimenez. An ambient intellgent language for dynamic adaptation. In *Workshop on Object Technology for Ambient Intelligence and Pervasive Computing (OT4AmI 2006)*, 2006.

[10] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] A. Hachichi, G. Thomas, C. Martin, B. Folliot, and S. Patarin. A generic language for dynamic adaptation. In *11th International European Parallel Processing Conference (EuroPar)*, 2005.

[12] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[13] A. Nicoara and G. Alonso. Dynamic aop with prose. In *International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA)*, pages 125–138, 2005.

[14] J. Paluska, H. Pham, U. Saif, M.Chau, C. Terman, and S.Ward. Technique-based programming. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, 2007.

[15] M. Roman and R. Campbell. Gaia: enabling active spaces. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop (EW)*, 2000.

[16] A. Senart, R. Cunningham, M. Bouroche, N. O'Connor, V. Reynolds, and V. Cahill. MoCoA: Customisable middleware for context-aware mobile applications. In *International Symposium on Distributed Objects and Applications (DOA)*, 2006.