Towards Context-Aware Propagators: Language Constructs for Context-Aware Adaptation Dependencies

Engineer Bainomugisha*, Wolfgang De Meuter, Theo D'Hondt
Programming Technology Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussels, Belgium
{ebainomu, wdmeuter, tjdhondt}@vub.ac.be

ABSTRACT

A context-aware system needs to reason about its current context of use and select applicable adaptations to activate or deactivate. This process is complex as often multiple contexts are available and improper interpretation of adaptation dependencies may lead to inconsistent or annoying system behaviour. This paper proposes a programming language support for defining context-aware dependencies between adaptations. Our model is based on the ideas of the *propagator* computational model to provide support for multiple dependencies that can coexist even if they contradict. Our proposed model is analogous to relationships and multiplicities in the modelling approaches. In addition, rather than fixed dependencies between adaptations, our model allows these dependencies to change depending on the context of use.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.3.3 [**Language Constructs and Features**]: Constraints; Modules, packages

Keywords

context-aware systems, context-oriented programming, context reasoning, dependencies, propagators

1. INTRODUCTION

Context-aware systems are able to adapt their behaviour depending on their context of use without explicit user intervention [1]. The context of use consists of context information such as current location, time, user preference, user's mood or surrounding situation. By context information we mean, any information that is computationally accessible [4]. Whenever there is context change, the system needs to reason about the current context and select the applicable adaptations to activate or deactivate. Adaptations may have dependencies between them to specify the semantics of how the system is adapted to the new behaviour. Managing adaptation dependencies is complex as often multiple contexts are available and improper interpretation may lead to inconsistent system behaviour [3]. This becomes even more complex when dependencies themselves may change, composed, or redefined depending on the current context of use. In such cases, multiple dependencies can coexist even if they contradict. The lack of explicit language support for expressing dependencies implies that the developers have no option but to entangle the logic of managing relationships between adaptations, as part of the application implementation.

This paper proposes a programming language support for expressing context-aware dependencies between adaptations. Our model is based on the ideas of the *propagator* computational model [6] to provide support for multiple dependencies that can coexist even if they contradict. This proposed model corresponds to relationships and multiplicities in modelling approaches such as [2, 5]. In addition, rather than fixed relationships between adaptations, our model makes dependencies among adaptations change depending on the context of use.

*Author funded by SAFE-IS project in the context of the Research Foundation - Flanders (FWO)

COP '09, Genova, Italy

2. MOTIVATING SCENARIO

This section presents a Context-aware Fire Sprinkler System scenario that we use to discuss the issues that apply to managing context-dependent adaptation dependencies.

2.1 Context-aware Fire Sprinkler System

Consider a hypothetical Context-aware Fire Sprinkler System (CaFSS) that provides fire fighting and preventive services. The system is fitted with sensor agents for detecting smoke, heat or emissions. The CaFSS classifies fire information (class A, B, C or D) and depending on this context information such as class of fire, the appropriate pipe (CO2, Foam, Water, or Powder) is selected and automatically discharged to smother the fire. The extinguishing speed varies depending on the place of installation (i.e. 100mm diameter for indoor, and maximum for outdoor). As a preventive measure the system sends an SMS to the firemen and triggers the alarm system in case the surroundings temperature goes above 50 degrees Celsius. Figure 1 presents an informal design representation of the CaFSS.

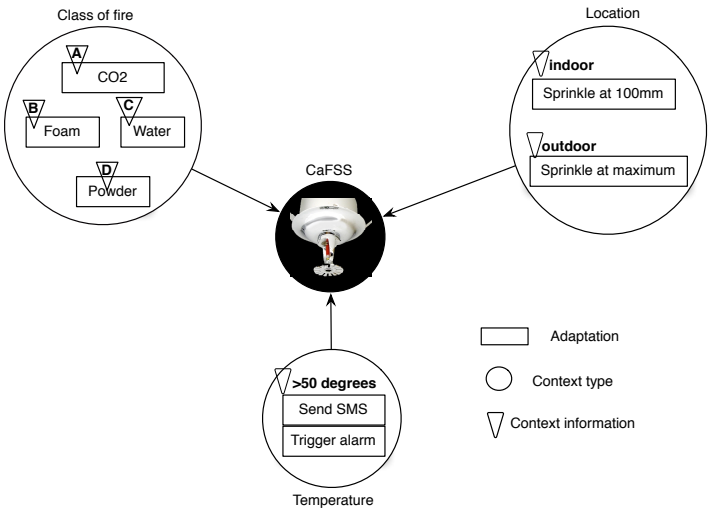

Figure 1: Scenario: Context-aware Fire Sprinkler System (CaFSS)

2.2 Scenario Analysis

Figure 1 shows the CaFSS that behaves differently depending on the context information. Context information is categorised as events (class of fire, temperature, and location), represented as circles. Inside the circle is the context information associated with its system behavioural adaptation (represented as rectangular boxes). For instance, when the fire of class A is sensed then the system is adapted to discharge CO_2.

To ensure that the system does not end up in an inconsistent state, proper determination of what adaptation (behaviour) is activated or deactivated for which class of fire (context) is very important. For instance, assume that both categories of fire class A and class B are sensed at the same time. In the default case this may imply discharging CO_2 and Foam extinguishers simultaneously. In this particular scenario, this may lead to a fatal situation as the mixture of the two chemicals may ignite the fire instead of extinguishing it. Therefore, developing a context-aware system requires the programmer to clearly implement how the system reasons about adaptation dependencies before activating the system behaviour that matches the current context of use.

2.3 Towards Context-Aware Dependencies

In this section we identify the issues that need to be dealt with in managing the dependencies amongst context-dependent adaptations.

Context-aware dependencies Dependencies define relationships among the context-dependent adaptations. These dependencies may also change depending on the context of use. Rather one fixed relationship between adaptations, new dependencies can be selected, defined or composed as the context changes. For example, in the CaFSS scenario, CO_2 and Foam may be incompatible when the system is indoor but may require each other for outdoor fires or when the temperature is below certain degrees.

Some of the modelling approaches already provide ways of expressing context information and relationships among

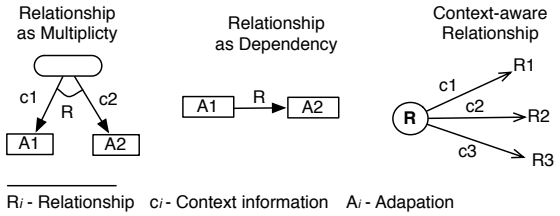

Figure 2: Context-aware Dependencies

adaptations. For example, Desmet et.al. propose Context-Oriented Domain (CODA) diagram [2] that extends Feature Diagrams [5] with context information. Figure 2 shows the CODA diagram with multiplicity relationship to specify how many adaptations are selected when contexts c1 and c2 are available. The rounded box represents context-independent behaviour that serves as a variation point for context-dependent adaptations (represented as rectangular boxes). The dependency relationship specifies the system semantics in case adaptations A1 and A2 are applicable at the same time. Rather than one fixed dependency, we argue that the relationship R should be be context-aware i.e. it can take on different forms R1, R2, or R3 depending on contexts c1, c2, or c3, respectively. This means that for A1 and A2 adaptations the dependency between them varies based on the current context of use. Therefore, the need for context-aware adaptation dependencies.

Dealing with contradictions amongst dependencies Context-aware dependencies imply that multiple dependencies may coexist with each other. However, the semantics of these dependencies may contradict each other and in such cases one adaptation dependency must be selected. Therefore, there is a need to deal with these contradictions. For example in the CaFSS scenario, providing support for context-aware dependency between the CO_2 and Foam adaptations. If one rule specifies that CO_2 and Foam can not coexist, and the second specifies that when temperature is below

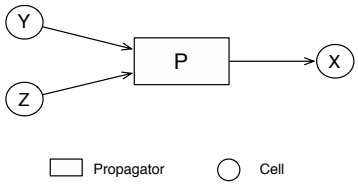

Figure 3: The Network of Propagators

20 degrees Celsius then CO_2 requires Foam.

Existing approaches such as Context-Oriented Domain (CODA) [2] attempt to solve some of these issues at the design level by introducing context information and relationships to the Feature Diagrams [5]. For example in CODA the exclusion dependency relationship implies that two adaptations can not be activated at the same time. The major limitation of this approach is that the dependencies and multiplicities are static in nature and are meant to remain the same throughout the system lifetime. To the best of our knowledge, we observe that no existing language support that addresses these issues.

The next section presents the propagator model in general that we use to address these issues. The concrete application of propagators to context-aware dependencies is explained in Section 4.

3. PROPAGATORS

In this section we discuss the main concepts of the *propagators* model proposed by Radul and Sussman in 2009 [6], that we use to address the issues identified in the previous section. This computational model differs from the conventional programming models in that it allows a variable to accept values from multiple sources. It is designed as network of *cells* and *propagators*. Cells correspond to the places that accept inputs from multiple sources and propagators continuously monitor the cells of interest and produce output to other cells. Propagators perform specified actions on its inputs, are asynchronous and stateless.

This computational metaphor implies no individual source is responsible for computing the value for the variable. We illustrate the propagator model with an example in Figure 3. The propagator P consumes values from Y and Z cells and writes its output to the cell X. The advantage of this model is that source of the final result needs not to be determined upfront. This means it is possible to build systems whose information flow depends on how they are used [6].

Allowing multiple sources for a single value comes with consequences such as determining which source to consider for the value. For example, for the cell X we must decide between Y and Z cells. A concrete practical example of using this propagation computational model is explained in [6] where the height of the building is calculated using two independent formulas that take on as inputs initial estimates. The propagation model proposes a number of techniques for dealing with the issue of determining the source to consider for the final result, that we discuss below:

Merging This is where results from multiple sources can be combined to yield an aggregate answer. For instance by taking an average for integer values or intersection for interval results. One limitation of this technique is that different sources need to wait for each other.

Multidirectional Computation This technique enables the refinement of the information contained in the cells. Once the final result is computed, information can propagate backwards to refine some of the initial inputs. For example, the final result of the height of the building calculated using two formulas can be used to improve the accuracy of the initial estimates.

Dependencies Decorations Rather than perform a merge on the multiple sources, a record justification ("decoration") is provided for each result. Cells can contain implementation of how to deal with information as it comes in from multiple sources based on the decorations. For instance, the initial estimates to the formulas of calculating the height of the building can be annotated with the value representing the error margin. By producing the height result from each formula along with extra information such as the error margin, the user can decide which final result to consider.

Our model is based on these techniques to determine the final dependency and to redefine dependencies based on context information. The next section presents our proposed *context-aware propagators* model.

4. CONTEXT-AWARE PROPAGATORS

We propose our conceptual model, *context-aware propagators* that is based on the idea of propagator computational model discussed in the previous section. In our model, a cell is designed to accept its inputs as context information, context-dependent adaptations, and dependencies. We propose two types propagators (1) context-adaptation mappings, which given context information outputs the corresponding adaptation(s) (2) Dependency definition, which given a context-dependent adaptations outputs the dependencies between these adaptations. Our model provides an interface to the COP language such as [4] as an output cell that contains the adaptations to (de)activate and dependencies between these adaptations. We refer to this cell as the *interface cell*.

As multiple adaptations become available, propagators need to define dependencies that are written to the interface cell to specify the semantics between adaptations. Providing support for context-aware dependencies implies that multiple dependency relationships can be available at the same time. Thanks to the propagator model, the fact that a variable can take on values from multiple sources enables us to design the interface cell to accommodate multiple adaptation dependencies. This means that multiple dependencies can coexist even if they contradict. In such case one dependency must be selected. We discuss the dependency selection techniques in the remainder of this section.

4.1 Propagator Based Dependency Selection

To address all these issues of deciding applicable adaptations and dependencies from multiple sources, we make use of the propagator model techniques as discussed below:

Merging The interface cell may hold extra information about the dependencies to perform the merging operation on adaptations or dependencies as they come in from multiple sources. For example in case of non-contradicting dependencies the merging operation can simply be a composition. Existing Context-Oriented Programming

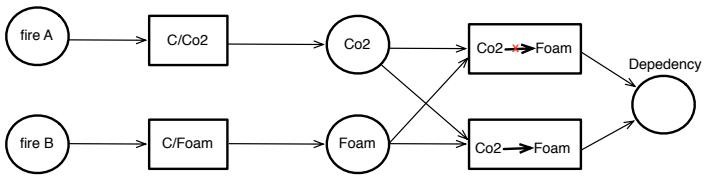

Figure 4: Context-aware propagators

(COP) languages such as ContextL [4] provide some support for composing adaptations.

Multidirectional Computation Using this technique, existing dependencies and adaptations can be dynamically updated by propagating backward the network. In our model we realise that backward propagation may need to be delimited to some scope to specify when and what cells that can be updated to avoid inconsistencies. For example in the CaFSS scenario it is not possible to overwrite fire.

Dependencies Decorations Using this technique, the propagator annotates each dependency information such as the dependencies that can coexist with each other, or the context when it becomes applicable. Then the interface cell can perform dependency selection based on this extra information. This may also be useful in cases where the user interaction is required. For example, by outputting the applicable dependencies and letting the user take the decision.

4.2 Example

To illustrate the concrete application context-aware propagators to adaptation selection, we consider the CaFSS scenario introduced in Section 2. Figure 4 shows the complete network of the propagator model that depicts the application of context-aware propagator to the CaFSS. The cells are represented as circles and contain context information, fire class A and fire class B. The propagators are represented as rectangular boxes. For example c/CO_2 propagator contains mappings between class of fire and the adaptation. This implies that when fire class B input is received, the c/CO_2 propagator outputs CO_2 adaptation. The cells CO_2 and Foam contain the actual adaptations which are also inputs to other propagators. The propagator CO_2-x->Foam defines an exclusion dependency between CO_2 and Foam adaptations, while the CO_2->foam propagator defines the an inclusion dependency between CO_2 and Foam adaptations. The final cell of this propagator is an interface to the actuator and contains the final dependency or set of adaptations to be activated or deactivated. Therefore, the decision of which adaptations to (de)activate is contained in the cell labelled dependency.

5. CONCLUSION AND FUTURE WORK

This paper focusses on language support for context-aware dependencies between adaptations. Our model aims at providing explicit means of expressing dependencies between adaptations, making sure that these dependencies are not always tied, and dealing with contractions in cases of multiple dependencies. We propose our ongoing work called *context-aware propagators* which is based on the propagator computational model. Using this model multiple dependencies between adaptations can coexist at the same time even if the contradict. Our model relies on multidirectional and dependency decorations techniques to deal with contractions and selecting the applicable dependencies.

This work is still in its early stages and we are currently investigating on what further refinements that need to be performed on design and implementation of the propagator model so as to provide support for context-aware dependencies. For example, we envision that the propagators multidirectional technique needs to include ways to limit the scope of backward propagation to avoid inconsistencies such as overwriting context information. In our future work we would like extend our model to work in a distributed setting where dependency selection is through harmonisation of cells and propagators on multiple hosts.

6. ACKNOWLEDGEMENTS

This work was partially funded by the Research Foundation - Flanders (FWO), the MoVES project, and the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB). The authors would also like to thank Jorge Vallejos and Pascal Costanza for the very fruitful discussions.

7. REFERENCES

[1] M. Baldauf and S. Dustdar. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, page 2004, 2004.

[2] B. Desmet, J. Vallejos, P. Costanza, and W. D. Meuter. Context-oriented domain analysis.

[3] J. Hähner, C. Becker, and P. J. Marrón. Consistent context management in mobile ad hoc networks. In *GI Jahrestagung (1)*, pages 308–313, 2004.

[4] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151–621, 2008.

[5] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[6] A. Radul and G. J. Sussman. The (abridged) art of the propagator. In *ILC 2009: Proceedings of the International Lisp Conference 2009*. ACM, 2009.

[7] J. Vallejos, P. Ebraert, B. Desmet, T. V. Cutsem, S. Mostinckx, and P. Costanza. The context-dependent role model. In J. Indulska and K. Raymond, editors, *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS '07)*, Lecture Notes in Computer Science, pages 277–299. Springer-Verlag, June 2007.