Uniting Global and Local Context Behavior with Context Petri Nets

Nicolás Cardozo[1,2], Sebastián González[1], Kim Mens[1], Theo D'Hondt[2]

[1]ICTEAM, Université catholique de Louvain
Place Sainte-Barbe 2 - 1348 Louvain-la-Neuve, Belgium
[2]Software Languages lab, Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels, Belgium
{nicolas.cardozo,s.gonzalez,kim.mens}@uclouvain.be, tjdhondt@vub.ac.be

ABSTRACT

Context-oriented programming enables adaptation of systems to their execution environment. Behavioral adaptations are defined in the system and then associated to a context. Such adaptations are made available at runtime when their context is deemed more appropriate by the execution environment. Context activation is reified using two techniques. Global to all running threads in the system, or local to a particular thread of execution. Providing one technique or the other may hinder the adaptable capabilities of the system. This paper extends the context Petri nets model to unify global and local context behavior. Global and local context behavior are represented as multicolored tokens in context Petri nets, by assigning a color to each thread in the system. By means of context Petri nets, context-oriented systems can unambiguously adapt their behavior globally, or to a particular thread of execution.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages

Keywords

Context-oriented programming, Consistency management, Behavioral adaptations, Petri nets

1. INTRODUCTION

Over the past few years, research in the field of Context-Oriented Programming (COP) [6] has focused on the implementation of languages and language extensions to enable dynamic adaptation to a system's execution environment. Nowadays, there are several COP languages which provide language abstractions for the definition of context, behavioral adaptations associated to a context, and the dynamic activation and deactivation of contexts [1]. However, the differences between such languages are marginal.

Our research vision builds on top of existing COP languages by focusing on the software engineering aspects of the development of context-aware systems. In particular we focus on providing tools and tool support for the development of such systems. This vision is motivated by the observation that behavior inconsistencies may exist in a system due to context changes in its execution environment. Behavior inconsistencies may emerge, for example, from the interaction between contexts or the unexpected deactivation of a context while the behavior it provides is being used.

To deal with behavior inconsistencies, we envision a COP system architecture that comprises all aspects in the life cycle of a COP system, ranging from the *context discovery* to the *system behavior* [8] and back. In particular, we use of a *context manager* module that arbitrates the interaction between contexts whenever they are to be activated. Contexts are effectively activated or deactivated only if such changes do not yield a behavior inconsistency. To manage contexts and their activations we use a Petri net-based model called context Petri nets.

context Petri nets (CoPN) is a programming model that ensures consistency in COP systems, which provides a live representation of contexts and their activations —that is, if a context can be activated, and how many times has it been activated. In this paper we take advantage of this property to reconcile one of the main differences between existing COP languages. Namely, *context activation*. The family of COP languages can be cleanly divided into two sets. On the one hand, there are languages for which a context activation has a *global* effect on the system. On the other hand, there are languages for which a context activation has a *local* effect to the current thread of execution in the system.

The use of one technique or the other depends on the requirements of the application domain. Some situations prefer a global context behavior over a local one. While others may prefer a local context behavior over a global one.

To provide both global and local context activations, the CoPN model is extended to accept multicolored tokens. Each

COP'12, June 11, 2012, Beijing, China

thread is identified with a particular color. Local activations on the thread only take place for that color. All global activations are identified with a single token color. Behavioral adaptations associated to a context are made available to all running threads in the system, or a particular thread, based on the token color with which a context is marked.

The reminder of this paper is organized as follows. Section 2 presents a motivating example for a unified model of context activation techniques. Our propose model for global and local activations is presented in Section 3 by first providing an introduction to the basic model and its programming interface, and then the describing the support for global and local context behavior. Related work is discussed in Section 4. Finally the paper is rounded up by the conclusion and future work in Section 5.

2. GLOBAL AND LOCAL CONTEXT BEHAVIOR

A *context* is defined as the reification of a situation computationally accessible in the execution environment of a system. Contexts are associated with particular behavioral adaptations that modify the application behavior at runtime.

The Context-Oriented Programming (COP) community proposes two visions of context. In the first vision, behavioral adaptations associated to a context are understood to influence the complete system. That is, *all* running threads in the system are conscious of *all* changes in the execution environment [10, 8]. These changes are said to be *global* to the system.

In the second vision, behavioral adaptations associated to a context are understood isolated. That is, changes in the execution environment of a system are only available within the dynamic scope in which the corresponding context was activated [6]. The behavioral adaptations associated to the context are not available for any other executing thread. These changes are said to be *local* to a thread.

We argue that the use of only one of these visions for context activation is too restrictive and hinders the adaptability of the system. On the one hand, only providing global context behavior restricts the application domains in which COP could be used. For example, if only a global context behavior is provided, it is not possible to have specific variations of a single application that changes dynamically with different users. On the other hand, if the system only provides support for local context behavior, different application threads may present different behavior according to the active contexts for the thread. Hence, not all threads would necessarily present the most appropriate behavior according to the execution environment.

2.1 Triangle Drawing Application

To illustrate the need to support global and local context behavior we use a small example application that consists on drawing moving triangles on a mobile device's screen. The triangle drawing application, comprises a canvas in which a set of triangles can be drawn. Triangle's behavior is given by a thread that autonomously moves each triangle forward in the canvas. The direction in which a single triangle moves can be modified to one of four directions `Up`, `Down`, `Left`, or `Right`, each of them represented by a context and specializing an abstract `Direction` context. The direction in which a triangle moves dictates the color of the triangle. A `Color` context is used in combination with the `Direction` context to give triangles an specific color. The application also allows to log the movement of a triangle in the canvas by the `Log` context. The direction, color, and logging of movement is a behavior that should be independent for each triangle.

Additionally, the application presents two behavioral adaptations that all existing triangles in the application should exhibit. A `Dashed` context draws all triangles with a dashed line, instead of a full line. A `Pause` context pauses the movement of all triangles.

In the triangle drawing application, though a proof-of-concept, is already possible to see the usefulness of providing global context behavior (e.g., `Pause` triangles movement) and local context behavior (e.g., `Log` a triangle's movement). If only one of the two techniques is used, the behavior of the other one would have to be manually supported.

3. UNITING GLOBAL AND LOCAL CONTEXT BEHAVIOR

This section presents our programming model to support global and local context behavior in COP systems, called context Petri nets (CoPN). CoPN is a runtime model for the consistent management of COP systems [5]. In particular, we use Subjective-C [8] as our base COP language.

3.1 Context Petri Nets

CoPN (read *co-pen*) is a Petri net-based runtime model for the management of COP systems. In particular CoPN uses three extensions of the Petri net formalism, namely, reactive Petri nets [7], static priorities [2], and inhibitor arcs [3]. The purpose of this paper is to present the extension of CoPN to support global and local context behavior in a COP language. A full formalization and motivation behind the CoPN model is provided as a technical report [4].

CoPN provides a concrete and live representation of a COP system by associating the concepts of COP to those of Petri nets. Figure 1 shows a prototypical example of a context (`Pause`) represented as a CoPN.

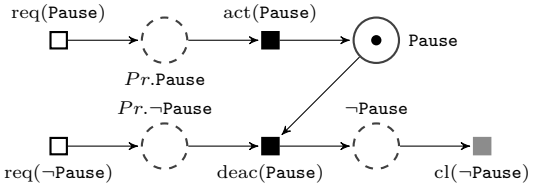

Figure 1: CoPN definition of the `Pause` context

Contexts are mapped to places in a Petri net. Actually, a context is an ensemble of 4 places which represent the state of the context as shown in Figure 1 for the `Pause` context. *Context places* (solid-line circles in Figure 1) represent a context being active. *Temporary*

places (dashed circles in Figure 1) respectively represent the states: prepare for activation ($Pr.$`Pause`), prepare for deactivation ($Pr.\neg$`Pause`), and already deactivated ($\neg$`Pause`).

Context state is mapped to Petri net tokens. Whenever a token resides in a place, this means that such state is currently valid. For example, in Figure 1 the context `Pause` is active because the place labeled[1] `Pause` is marked by a token. Respectively, the context is preparing for activation if place $Pr.$`Pause` contains a token, preparing for deactivation if place $Pr.\neg$`Pause` contains a token, and already deactivated if place $\neg$`Pause` contains a token.

Context activation/deactivation is mapped to Petri net transitions. Intuitively transitions with an outgoing arc to a place, are activations for the context state represented by the place. Transitions with an incoming arc from a place, are deactivations of the context state represented by the place. Figure 1 exhibits three different types of transitions. *External transitions* (white squares in Figure 1), which are triggered as a consequence of a context change in the system's execution environment. *Internal transitions* (black squares in Figure 1), fire automatically whenever they become enabled. *Internal cleaning transitions* (gray squares in Figure 1), fire whenever they are enabled, if none of the black transitions is enabled.

Transition colors define the priority in which transitions must fire. Higher priority transitions always fire before lower priority ones. Black transitions have the highest priority, and white transitions have the lowest priority.

Tokens flow in a Petri net by means of transition firing. In order to fire, a transition must be enabled. In CoPN transitions are enabled if: their input places (i.e., incoming arcs to the transition) from normal arcs ($\rightarrow$) are marked, their input places from inhibitor arcs ($\multimap$) are not marked, and no other transition with a higher priority is enabled. Once fired, tokens flow from the input places to the output places.

Figure 2 shows a CoPN with initial marking m_0(`D`)=1 and m_0(`C`)=2 for the `Direction` (`D`) and `Color` (`C`) contexts in the triangles application. Suppose transition $req.(\neg$`D`$)$ is fired, yielding marking m_1 which is given by m_1(`D`)=1, m_1(`C`)=2, $m_1(Pr.\neg$`D`$)$=1. In this configuration the only enabled transition is $deac($`D`$)$. Note that, although the external transitions fulfill the requirements about their inputs, they are not enabled as they have a lower priority than $deac($`D`$)$. Firing the transition removes a token from the respective input places $Pr.\neg$`D` and `D`, and adds a token to the $\neg$`D` place. The firing enables the $deac($`C`$)$ transition to the right of the `C` place. Firing this transition two times removes both tokens from place `C`. After this, the only enabled transition is the gray transitions $cl(\neg$`D`$)$. After the transition fires, the CoPN yields an empty marking.

[1]Labels in CoPN are mere decorations for places and transitions and have no semantic meaning in the model.

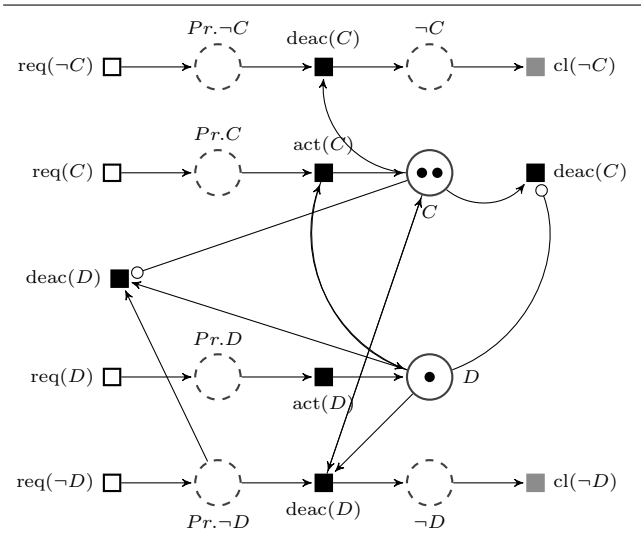

Figure 2: CoPN transition enabling and firing

3.2 Context Petri nets in Subjective-C

This section presents the language interface to interact with the CoPN model in Subjective-C,[2] a COP language extension to Objective-C. Additional to the basic COP language constructs, Subjective-C has an integrated *context manager* module. The context manager is responsible for arbitrating the interaction of context activation and deactivation. To ensure that context changes do not lead to inconsistencies, interaction between contexts is managed by means of dependency relations.

The CoPN model plays the role of the context manager in our Subjective-C implementation, arbitrating the activation of context behavior and ensuring its consistency. The use of CoPNs is transparent to developers since they interact with it indirectly through Subjective-C's language abstractions.

CoPN supports the definition of contexts, their dynamic activation and deactivation, behavioral adaptations definition, and contexts composition. Contexts are defined by means of the `@context` construct. The `Pause` context given in Figure 1 is created as `@context(Pause)`. Interaction between contexts can be defined by means of dependency relations. Currently, Subjective-C supports 4 kinds of dependency relations *exclusion*, *weak inclusion*, *strong inclusion* and *requirement* [8], however, other dependency relations could be defined. A dependency relation is defined by giving the type of the dependency and the contexts involved in the interaction. For example, Figure 2 shows a requirement relation between contexts `C` and `D` which is defined as `[addRequirementOf: D to: C]`. Once the CoPN is created, a context can be activated or deactivated by means of the `@activate` and `@deactivate` constructs, respectively. Subjective-C uses timestamps to keep track of the latest activation of a context. Whenever a method is defined for two active contexts,

[2]A full implementation of context Petri nets is currently integrated with Subjective-C and is available for download at `http://released.info.ucl.ac.be/Tools/Context-PetriNets`.

the visible behavior will be that of the context activated the latest.

In CoPNs contexts are not immediately activated. Instead, context activation is requested by firing an external transition, e.g., *req*(`Pause`). Such firing may enable some of the internal transitions which automatically fire. If only context places are marked after all enabled internal transitions have fire, the context is effectively activated. If one of the temporary places is marked, changes in the CoPN marking triggered by the activation request (*req*(`Pause`)) are reverted to the marking before the external transition firing. In such a case, the CoPN model presents the user with feedback about the cause of the activation failure. Context deactivation follows a similar process to that of context activation.

The usefulness of using CoPN as the runtime model for the consistency management of the system lies in that all information regarding whether or not a context (de)activation yields to an inconsistency, is directly available in the model.

3.3 Supporting Global and Local Context Behavior

Up to this point we have only discussed how COP systems are structured and managed by CoPNs. All context activations and deactivations discussed so far are *global* contexts activations, as this is the technique used in Subjective-C.

To introduce *local contexts* (i.e., contexts that are specific to a thread of execution) we take advantage of the thread model provided by Objective-C, and Colored Petri nets [11], a Petri net extension to model concurrent processes. Colored Petri nets extend the basic Petri net model by adding colors to tokens. Thus transitions may be enabled for a particular color. Transition firing is modified to take into account the color of input and output tokens.

Providing a local activation of contexts implies that a context and its associated behavior adaptations will only be available in the threads for which the context is active. All other threads in the system are oblivious to the context.

At the language level, the `@activate` and `@deactivate` constructs are modified to take into account the thread in which the action is to take place. Figure 3 shows the activation of the `Log` context in an specific thread, $thread_{blue}$. The activation is expressed as `@activate(Log in` $thread_{blue}$`)`. In CoPN such an activation is represented by a colored token. Each thread in the system is identified with a unique color. Activation of the `Log` context in the $thread_{blue}$ thread is then represented by the introduction of a *blue* token. This is shown in Figure 3 by the introduction of a blue token in the *Pr*.`Log` temporary place. Note that the `Log` context is already active in two other threads, $thread_{green}$ and $thread_{red}$ as the `Log` place is marked by green and red tokens. Color of tokens is not modified by transitions. The firing of the *act*(`Log`) transition moves the blue token to the `Log` place.

In CoPN global context activations are represented by *black* tokens. Unlike local context activations, behavioral adaptations of contexts marked with a black token are available for *all* running threads in the system.

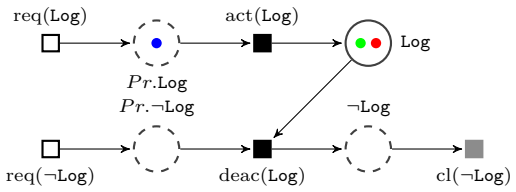

Figure 3: Three thread local activations for the `Log` context

```
//thread creation for the t1 triangle
NSThread *thread_green = ...
//thread creation for the t2 triangle
NSThread *thread_blue = ...
//Triangles start moving
//movement of first triangle is logged
@activate(Log in thread_green);
//all triangles drawn with dashed lines
@activate(Dashed);
```

Snippet 1: Effect of global and local context activations

Snippet 1 shows a piece of code for the triangle drawing application that exemplifies how global and local contexts are used. In the snippet we assume to have two triangles t_1 and t_2, each of them associated to an independent thread, $thread_{green}$ and $thread_{blue}$ respectively.

Triangles start to move independently based on their direction. Suppose we decide to log the movements for triangle t_1. This is represented by the call to `@activate(Log in` $thread_{green}$`)`. After the context has been activated (supposing the activation does not yield any inconsistencies) all movements of t_1 will be logged. However, the movements of triangle t_2 remain unlogged. Finally, the `@activate(Dashed)` message makes available the method to draw triangles in a dashed line. As no thread is specified in this activation, all triangles are drawn with a dashed line. Note that a thread could have been given to the `Dashed` context. CoPN allows to activate any context globally or locally.

Global and local context behavior interaction exists only when contexts interact with each other via a dependency relation. Intuitively if the conditions that reified a context are valid globally, they are also valid for a particular thread. If such context conditions are not longer valid globally, they are not valid for any thread. This interaction between global and local context activations can be better explained using the color theory analogy of color subtractive composition. In color theory, the black color can be seen as the subtractive combination of all colors i.e., it is the result of mixing all colors. In CoPN global context activations (black tokens) can be seen as activations in every thread in the system (containing all colors), hence contexts activated in particular threads, can depend or be influenced by global context activations. When a context activation depends on, or influences other contexts through a dependency relation, the interaction takes into account the thread of the activation.

If the activation of a context depends on a second context, the latter context can be active both globally, or in the same

thread the former context is to be activated in. If the deactivation of a context generates de deactivation of a second context and the former context is being deactivated globally, the latter context is also deactivated even if its active in an specific thread.

Figure 4 revisits the requirement relation between the `C` and `D` contexts of Figure 2, showing the interaction of global and local context activations. Take as the initial marking of the CoPN m_0 to be m_0(D)=1. Suppose that context `C` is to be activated in an specific thread $thread_{red}$ by the construct `@activate(C in` $thread_{red}$`)`. After the firing of the req(C) transition, the marking of the CoPN is m_1 where m_1(D)=1 m_1(C)=1_{red}.[3] Note that the act(C) transition was fired because the input place `D` is marked with a black token and, by subtractive composition, red.

Similarly, if context `D` is deactivated by the construct `@deactivate(D)` the firing of $req(\neg$D) will lead to a marking m_2 where m_2(D)=1, $m_2(Pr.\neg$D)=1, and m_2(C)=1_{red}. With this marking the bottom most $deac$(D) transition can fire. The firing of the transition leads to marking m_3 where $m_3(\neg$D)=1, and m_3(C)=1_{red}. Note that transition $deac$(C) can fire even though the deactivation is being performed globally. The firing of such transition and the enabled internal cleaning transition $deac(\neg$D) leads to an empty marking.

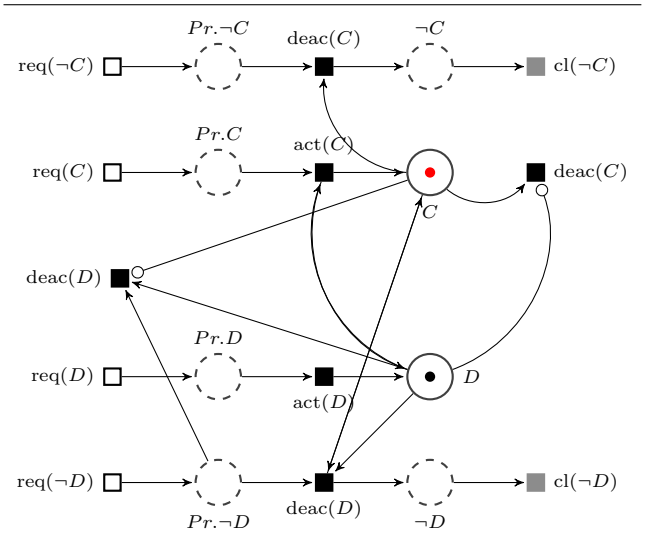

Figure 4: Global and local context activation interaction

4. RELATED WORK

Global context behavior is supported in Ambience [9] and Subjective-C [8]. Both Ambience and Subjective-C maintain a representation of all active contexts in the system called the *current context*. At runtime, all threads in the system gather the information about active contexts from the current context. Whenever a context is activated or deactivated, the current context is modified by respectively adding or withdrawing contexts from it. Hence, such actions influence all running threads. ContextErlang [14] is a COP language that supports distribution an concurrency natively. In ContextErlang, active contexts are registered in a *variations stack* per distributed component in the system. Thus, context activations and deactivations are applied to all threads running in such a component. Lambic [15] is COP language with explicit distribution. In Lambic behavioral adaptations are defined as predicate methods, a method with an associated predicate describing its context conditions. Behavioral adaptations become available in the system, if their predicate is valid. Predicates are evaluated every time a method is called. This means that as long as a predicate is valid, its corresponding behavioral adaptation would be seen by the complete system.

Local context behavior is supported by ContextL [6] and other layer based COP implementations [1]. These languages restrict context availability to the dynamic scope of the `with` construct for context activation. This means that whichever thread a context is activated in, is the only thread for which the context's associated behavioral adaptations are available. Other threads are oblivious to any context modifications. EventCJ [12] is an event-based COP language. In EventCJ contexts are not defined based on the dynamic scope of the context activation construct, but rather contexts can be activated for a particular object instance. Although EventCJ's context activation differs from other COP languages, context activations can be regarded as local per object instance.

Thus far, support for global and local context behavior has only been provided by ContextJS [13]. ContextJS is a layered based context-oriented extension of JavaScript. In ContextJS layers can be activated in the dynamic scope of the activation construct `withLayer`, or in the global scope of the system by explicitly activating the layer with the `enableLayer` construct. Globally activated layers remain active unless explicitly deactivated using the `disableLayer` construct. By default global behavior takes precedence over local behavior. However, ContextJS provides an open model which allows to define the activation scope of a layer. In particular the precedence of global and local behavior can be modified.

5. CONCLUSIONS AND FUTURE WORK

This paper presents an extension to the context Petri nets model (CoPN) by adding support for multicolored tokens. The purpose of this extension is to provide a unified support for global and local context behavior and their interaction in context-oriented programming (COP) systems. To do so, we extended the implementation of Subjective-C that reifies CoPNs, by allowing the activation and deactivation language constructs to specify in which thread a context is to be (de)activated. The CoPN model maps each of the different running threads in the system with a particular token color. When a context is activated in a particular thread, the context place is marked with a token of the respective color. Behavioral adaptations associated to the context are only made available in such thread, and oblivious to all other threads. Global context behavior is treated as an special case of the local context behavior. When a context is activated without specifying any thread, its activation is marked with an special token color (e.g., *black*). Whenever a context

[3]The notation 1_{red} is used to express that there is 1 red token in the place, if no color is specified we assume the token to be black.

is marked with a *black* token, its behavioral adaptations are made available to all running threads in the system. Interaction between global and local activations is automatically managed by the CoPN model.

Providing both global and local techniques of context activation by means of CoPN allows COP systems to present more flexible kinds of adaptations according to the requirements of the application domain.

Unifying the global and local context behavior is already a step forward into providing full adaptable systems with COP, however, we see interesting directions in which this work could be extended. An interesting direction for future work would be not to allow local context activations in a per-thread basis, but instead to reify local context activations in a per-object instance basis as in EventCJ. Note that to allow per-object instance local context activation instead of per-thread local context activation the CoPN model does not need to be modified.

Acknowledgements

This work has been supported by the ICT Impulse Programme of the Brussels Institute for Research and Innovation.We thank the anonymous reviewers for their comments on an earlier version of this paper.

6. REFERENCES

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM Press.

[2] F. Bause. On the analysis of petri nets with static priorities. In *Acta Informatica*, volume 33, pages 669 – 685, 1996.

[3] E. Best and M. Koutny. Petri net semantics of priority systems. *Theoretical Computer Science*, 96:175–215, April 1992.

[4] N. Cardozo, S. González, K. Mens, and T. D'Hondt. Context petri nets: Definition and manipulation. Technical report, Université catholique de Louvain and Vrije Universiteit Brussel, April 2012. `http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-07.pdf`.

[5] N. Cardozo, J. Vallejos, S. González, K. Mens, and T. D'Hondt. Context petri nets: Enabling consistent composition of context-dependent behavior. In *International Workshop on Petri Nets and Software Engineering*, PNSE'12, June 2012. (to appear).

[6] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, Oct. 2005. Co-located with OOPSLA'05.

[7] R. Eshuis and J. Dehnert. Reactive petri nets for workflow modeling. In *Application and Theory of Petri Nets 2003*, pages 296–315. Springer, 2003.

[8] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-C: Bringing context to mobile platform programming. In *Proceedings of the International Conference on Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 246–265. Springer-Verlag, 2011.

[9] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

[10] S. GonzÃąlez, K. Mens, and P. Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the Dynamic Languages Symposium*, pages 77–88, New York, NY, USA, Oct. 2007. ACM Press. Co-located with OOPSLA'07.

[11] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modeling and validation of Concurrent Systems*. Springer-Verlag, 2009.

[12] T. Kamina, T. Aotani, and H. Masuhara. Eventcj: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, AOSD'11, pages 253–264. ACM Press, Mar. 2011.

[13] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Sci. Comput. Program.*, 76(12):1194–1209, Dec. 2011.

[14] G. Salvaneschi, C. Ghezzi, and M. Pradella. Contexterlang: introducing context-oriented programming in the actor model. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 191–202, New York, NY, USA, 2012. ACM.

[15] J. Vallejos, S. González, P. Costanza, W. De Meuter, T. D'Hondt, and K. Mens. Predicated generic functions: Enabling context-dependent method dispatch. In B. Baudry and E. Wohlstadter, editors, *Software Composition*, volume 6144 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 2010.