Context Petri Nets

Enabling Consistent Composition of Context-Dependent Behavior*

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

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

Abstract. Ensuring the consistent composition of context-dependent behavior is a major challenge in context-aware systems. Developers have to manually identify and validate existing interactions between behavioral adaptations, which is far from trivial. This paper presents a run-time model for the consistency management of context-dependent behavior, called context Petri nets. Context Petri nets provide a concrete representation of the execution context of a system, in which it is possible to represent the interactions due to dynamic and concurrent context changes. In addition, our model allows the definition of dependency relations between contexts, which are internally managed to avoid inconsistencies. We have successfully integrated context Petri nets with Subjective-C, a context-oriented programming language. We show how our model can be cleanly combined with the abstractions of the language to define and manage context-dependent behavior.

1 Introduction

Current sensing technology allows computing devices to be highly aware of their execution environment. To leverage the full potential of these sensing capacities, software systems should properly represent the sensed context and dynamically adapt their behavior accordingly. To develop such systems, the Context-Oriented Programming (COP) paradigm has emerged [4], which enables the definition and composition of context-dependent behavioral adaptations. However, consistently composing behavioral adaptations is still challenging. Developers need to manually ensure that insertion and withdrawal of adaptations preserve the expected behavior of the system. Different approaches have been proposed to prevent inconsistencies by defining *dependency relations* between contexts and

* This work has been supported by the ICT Impulse Programme of the Brussels Institute for Research and Innovation.

their associated behavioral adaptations [6,8]. These dependencies constrain context interaction by conditioning the deployment of behavioral adaptations at a high abstraction level, which is well-suited to developers. Nevertheless, developers still have to manually check consistency of such interactions, which is far from trivial.

We claim that inconsistencies in the composition of context-dependent behavioral adaptations arise mainly by the *multiple* and *dynamic* nature of the system's context (an heterogenous collection of data which can vary dynamically over time, and from one location to another). Without the appropriate support to represent such context and to deal with their dependencies and dynamic changes, it is often difficult to ensure that the behavioral adaptations associated to them do not interfere with each other. We then propose a Petri net-based execution model for context-oriented programming, called context Petri nets (CoPN), which enable a consistent representation and management of the context of a system. In our model, context changes are modeled as dynamic context activations and deactivations. Dependency relations between contexts are expressed by connecting activation/deactivation actions of different contexts. In addition, context Petri nets provide a concrete view of the system's state at every point in time, easing consistency management. Every activation/deactivation that generates an inconsistent state is immediately retracted to the state before its execution.

The reminder of the paper is organized as follows. Section 2 gives a brief background on COP, putting forward the requirements to provide consistent composition of behavioral adaptations. Section 3 presents the foundations of our context Petri nets model, and Section 4 explains how this model fulfills the composition requirements. Section 5 and 6 assess the approach, its relation to existing work, and possibilities for future work. Section 7 concludes the paper.

2 Requirements for Consistent Composition of Context-Dependent Behavior

Context-Oriented Programming (COP) allows software systems to adapt their behavior dynamically according to changes detected in their execution environment [4]. The core characterization of COP systems from which we start comprises the following concepts:

- *Contexts* represent particular situations detected during the execution of an application, with respect to which application behavior can be adapted as deemed appropriate.
- *Context activation* takes place whenever the situation for which the context stands is detected in the execution environment; correspondingly, *context deactivation* takes place when the given situation no longer occurs in the execution environment.

COP allows systems to define behavioral adaptations which are associated to particular contexts. Therefore, the adaptations are dynamically composed

with the system's basic functionality whenever the contexts become active. As illustration of contexts and context activations, consider the case of a context-aware mobile phone with Internet connectivity. The phone can gain access to the Internet by means of three different technologies: `WiFi`, `3G` and `Edge`. These contexts are active whenever the respective protocol is available, and inactive otherwise. The fact that the phone has any connectivity at all is signaled by the activation of a `Connection` context. Such activation follows the activation of `WiFi`, `3G` or `Edge`. Besides Internet connectivity, the phone also supports video calls. When `Connection` is active, video calls become possible, a situation that is signaled by activating the `VideoCall` context. Video calls require that there is enough battery power left —that is, for the `HighBattery` context to be active.

Although simple, this scenario already shows two peculiarities of contexts and context activation:

Dynamicity The activation state of contexts changes unannounced over time as different situations are detected in the execution environment.

Multiplicity Multiple contexts can be active at the same time, including the case that a same context can be activated more than once.

As an example of dynamicity, the `WiFi` context can be active intermittently as the user roams around in the city and wireless networks are found and left behind. As for multiplicity, the `Connection` context can be activated as much as three times, depending on whether `WiFi`, `3G` or `Edge` are available.[3]

The dynamicity and multiplicity of context activation can compromise the behavioral consistency of COP systems. For instance, inconsistencies can arise if adapted behavior is withdrawn from the system while it is executing [10] —a consequence of dynamicity. Inconsistencies can also arise when the adaptations of an active context contradict the adaptations of another active context —a consequence of multiplicity. Hence, to ensure consistent composition of behavioral adaptations, a COP system should provide support to cope with dynamicity and multiplicity. This means that the following requirements should be fulfilled:

R.1 Dynamic Context Activation and Deactivation Provide a consistent representation of the system's context. This implies that dynamic context changes should be clearly reflected in the system as they are detected in the system's execution environment.

R.2 Consistent Interaction Between Multiple Contexts Ensure that programs are always in a consistent state, even after a context activation or deactivation. In case of multiple activations of different contexts, the model should prevent contexts that interfere with each other to be active at the same time.

R.3 Multiple Activations of the Same Context Allow that a context is activated as many times as different instances of the situation represented by the context actually occur in the execution environment.

At present, we observe that no single COP approach appropriately support these three requirements. Most COP languages [4,8,9,14] define dedicated constructs for context *activation* and *deactivation*. However, only few of them

[3] The concept of multiple context activation is analogous to that of multisets in mathematics, in which an element can appear more than once in the multiset.

provide means to specify constraints between contexts. Subjective-C [8] defines language abstractions to specify dependency relations which are internally represented and managed using a dependency graph. ContextL [6] and EventCJ [14] allow defining context interactions programmatically by means of transition functions. The main problem with these approaches is that they require that developers manually verify the consistency of the context dependencies. This means that they need to check every possible interaction between context (de)activations. Furthermore, these approaches do not provide a structured way to compose context dependencies. As a result of this, developers have to manually encode the composition which is cumbersome, error-prone and typically leads to programs that are difficult to understand and maintain.

Concerning the multiple context activations, ContextL, EventCJ, and other COP languages that follow similar design decisions allow contexts to be activated only once. Subjective-C and Ambience [9], on the other hand, allow contexts to be activated as many times as necessary.

We now proceed to explain our proposal to address these requirements using a formal tool from the realm of concurrent systems modeling.

3 Context Petri Nets

To ensure the consistent composition of behavioral adaptations, we introduce a run-time model for COP called *context Petri nets (CoPN)*.[4] CoPN (read *copen*) is a Petri net-based formalism based on three variants of Petri nets: reactive Petri nets [7], static priorities [1], and inhibitor arcs [2]. Petri nets have been used extensively to describe the information control flow of non-deterministic, concurrent systems. This makes such a formalism suitable to cope with the dynamicity and multiplicity of context in software systems. In this section, we explain how to use CoPN to model contexts, dependencies between contexts, and the composition between such dependencies. We then discuss how the execution semantics of our model ensures that contexts can be always consistently activated.

3.1 Structure of CoPNs

The CoPN model follows the definition of reactive Petri nets with inhibitor arcs and static priorities shown in Table 1. The components of a CoPN are defined by the tuple $\mathcal{P} =< P, T, f, f_\circ, \rho, m_0 >$ (1), where P is a finite set of places, T is a finite set of transitions, f is the flow function defining regular *arcs* between places and transitions, $f_\circ$ is the flow function defining *inhibitor arcs* between places and transitions, ρ is a function defining *priorities* of transitions, and m_0 is the initial marking function assigning *tokens* to places. This description of CoPNs follows from their formal definition [3].

[4] CoPNs are fully implemented as a run-time model for the Subjective-C [8] language. The implementation is available for download at `http://released.info.ucl.ac.be/Tools/Context-PetriNets`.

(1)	$\mathcal{P} =< P, T, f, f_\circ, \rho, m_0 >$	(5)	$f : (P \times T) \cup (T \times P) \longrightarrow \mathbb{Z}^+$
(2)	$P \cap T = \phi$	(6)	$f_\circ : P \times T \longrightarrow \{0, 1\}$
(3)	$P = P_c \cup P_t$	(7)	$\rho : T \longrightarrow \mathbb{Z}^+$
(4)	$T = T_e \cup T_i \cup T_c$	(8)	$m_0 : P \longrightarrow \mathbb{Z}^+$

Table 1: Context Petri nets components definition.

Places and transitions are disjoint sets (2). The set of places is divided into two disjoint sets: P_c of *context places*, and P_t of *temporary places* (3). The set of transitions is divided into three disjoints sets: T_e of *external transitions*, T_i of *internal transitions* and T_c of *internal cleaning transitions* (4). There cannot be arcs between two places or two transitions. Each arc defines how many tokens flow from, or to places (5). There can be maximum one inhibitor arc between a place and a transition (6). Transitions are given a firing order priority. Higher priority transitions fire before lower priority ones (7). Enabled transitions of the same priority fire randomly. Finally, tokens are assigned to places by means of the (initial) marking function (8).

An explanation of the mapping between Petri nets and COP concepts follows. As illustration, Fig. 1 shows how the `VideoCall` context from the example in Section 2 can be defined as a CoPN.

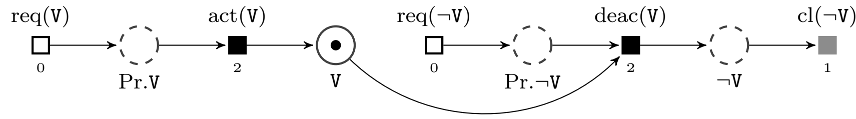

Fig. 1: CoPN representation of the `VideoCall` (`V`) context.

Places in CoPNs are used to capture the state of contexts. A context is defined in terms of four places defining the context's life cycle. A *context place*, P_c, (solid-line circle labeled `VideoCall` in Fig. 1) is used to represent the actual context and its activation state. The other three *temporary places*, P_t, (dashed circles in Fig. 1) are used to represent intermediate states of the context: preparing for activation (Pr.`VideoCall`), preparing for deactivation (Pr.¬`VideoCall`), and flagged as already deactivated (¬`VideoCall`).

Temporary places help to maintain consistency constraints when manipulating the activation state of contexts. Activation and deactivation of a context does not occur immediately, but needs to be requested first and processed carefully, since the request may be denied if it violates constraints imposed by other contexts. The *flag* temporary place (¬`VideoCall` in Fig. 1) is used to ensure that a context is effectively deactivated once for every deactivation request (otherwise, the context would be emptied of all its tokens after just a single deactivation).

Transitions in CoPNs represent changes in the activation state of contexts. Transitions are divided in two categories: external and internal. *External transitions* (white squares in Fig. 1) are used to *request* a context activation or deactivation in response to a change detected in the execution environment. *Internal transitions* (black squares in Fig. 1) forward the requests to other dependent contexts, and trigger the actual activation or deactivation of contexts. Finally, a particular kind of internal transition *internal cleaning transitions* (gray square in Fig. 1) is used to *clean* the deactivation flag place.

Transition priorities are shown as small numbers under each transition in Fig. 1. External transitions are white transitions of priority 0. Internal transitions are black transitions of priority 2. Internal cleaning transitions are gray transitions of priority 1. Transition priorities are unequivocally identified by the transition color, hence priorities will be omitted in future.

Tokens represent the activation state of a context, depending on the place they occupy. In Fig. 1 the `VideoCall` context is active if its context place (labeled `VideoCall`) is marked, preparing for activation if place Pr.`VideoCall` is marked, preparing for deactivation if place Pr.¬`VideoCall` is marked, and already deactivated if place ¬`VideoCall` is marked.

Arcs encode the possible ways in which tokens can flow from one place to another, mediated by transitions. Hence, arcs help encoding the way context activations and deactivations depend on each other. *Regular arcs*, noted as arrow-headed edges ($\rightarrow$), permit to verify the presence of tokens in a place, thanks to the f flow function. *Inhibitor arcs*, depicted as circle-ended edges ($\multimap$), permit to verify the absence of tokens in a place, by means of the $f_\circ$ flow function. Inhibitors are used for example to express that a context cannot be activated if another context is active.

3.2 Dynamics of CoPNs

CoPNs make it possible to represent and track the changes that occur in the system's execution environment. CoPNs can thereby be used as run-time representation of context. The following descriptions define the way context state is encoded in a CoPN, and how it evolves according to the constraints encoded in the structure of such CoPN.

- A transition t is *enabled* if its input places p_i from regular arcs contain at least $f(p_i, t)$ tokens, its input places $p_\circ$ from inhibitor arcs are empty, and no other transition t' with higher priority, $\rho(t') > \rho(t)$, is enabled.
- Transition *firing* modifies the state of the Petri net by removing as many as $f(p_i, t)$ tokens from its input places p_i, and adding as many as $f(t, p_{out})$ tokens to its output places p_{out}.
- External transitions are fired with the regular *may* fire semantics of Petri nets. That is, if a transition is enabled it may fire. In our model external transitions are fired as consequence of a change in the execution environment.
- Internal transitions are fired with a *must* fire semantics. That is, if an internal transition is enabled it must fire. Internal transitions are used to coordinate activation and deactivation among different contexts, according to the dependency relations established between them. Section 3.3 describes such dependencies.

CoPN model is used to ensure consistency of context activations, we define a CoPN to be in a *consistent state* if no temporary place is marked after *all* enabled internal transitions have fired.

3.3 Dependency Relations Between CoPNs

CoPN allows multiple activations of different contexts. To avoid conflicts in the adaptations of the different active contexts, our model enables the definition of dependency relations between contexts. We have taken as starting point the four dependency relations defined in Subjective-C [8], and modeled them in CoPN: *exclusion*, *weak inclusion*, *strong inclusion* and *requirement*. Each dependency relation defines how the activation state of a context influences that of another context. In CoPNs, this is achieved by connecting internal transitions of one context to the (temporary) places of another one, via an arc. Each arc expresses a *rule* describing the interaction between contexts. New dependency relations could be defined by describing such rules.

Exclusion. An exclusion dependency prevents two contexts from being active at the same time. However, both contexts may be simultaneously inactive. For example, the interaction between the `LowBattery` (`L`) and `HighBattery` (`H`) contexts of the mobile phone is defined by the CoPN shown in Fig. 2. These contexts clearly should not be active at the same time. If one of the contexts is active, the activation of the other is prevented by the corresponding inhibitor arc.

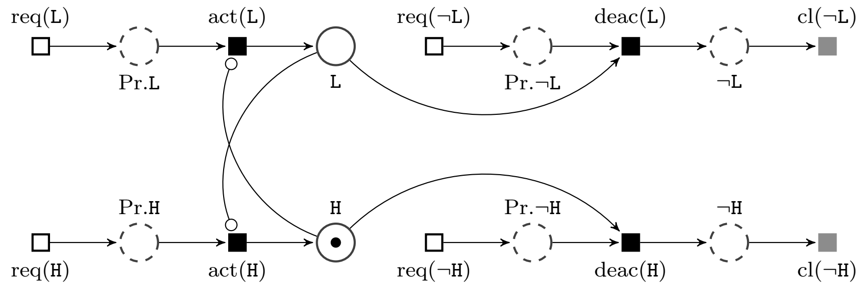

Fig. 2: Exclusion between `Low Battery` (`L`) and `HighBattery` (`H`).

Firing a request for activating the `L` context, $req(\texttt{L})$, under the initial marking $m_0(\texttt{H})$=1 yields the marking m_1, where $m_1(\texttt{H})$ =1 and $m_1(Pr.\texttt{L})$=1. At this point, none of the internal transitions is enabled. In particular, $act(\texttt{L})$ is not enabled because of the inhibitor arc $(\texttt{H}, act(\texttt{L}))$. An inconsistent state has been reached since one of the temporary places, $Pr.\texttt{L}$, is marked. In this case, all of the the actions are reverted to the initial marking state. The request for the activation is denied, and the user is informed about the reason for the refusing the activation.

Weak Inclusion A weak inclusion represents a situation in which the activation (deactivation) of a context should automatically trigger the activation (deactivation) of another context. Note that the latter context can be activated or deactivated independently of (without effect on) the former. This interaction is shown in Fig. 3, using as example the case of the `Connectivity` and `VideoCall` contexts: activation of `Connectivity` automatically triggers the activation of `VideoCall`, meaning that video calls are normally available whenever the phone is connected to the Internet). The double arc in Fig. 3 is a visual shortcut that stands for two different arcs going in opposite directions.

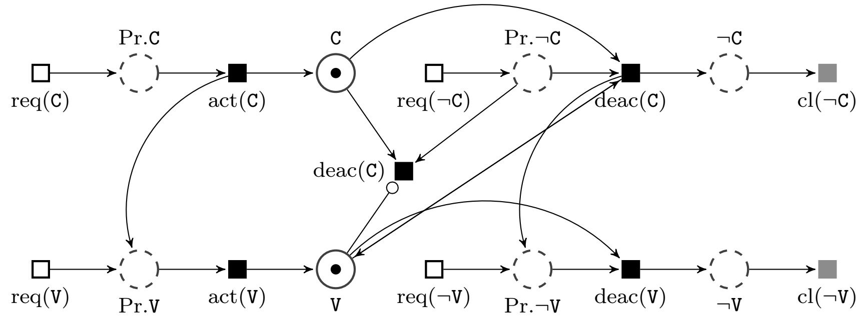

Fig. 3: Weak inclusion from `Connectivity` (`C`) to `VideoCall` (`V`).

Strong Inclusion A strong inclusion represents a dependency in which, similarly to a weak inclusion, activation or deactivation of a context triggers that of the related context. Additionally, deactivation of the latter context triggers back the deactivation of the former. These interactions are encoded by the CoPN shown in Fig. 4; as in weak inclusion, the double arc stands for two different arcs going in opposite directions. The CoPN encodes an interaction such that activation of `WiFi` results in the activation of `Connectivity`; reciprocally, if for some reason `Connectivity` is deactivated, then `WiFi` will also be deactivated.

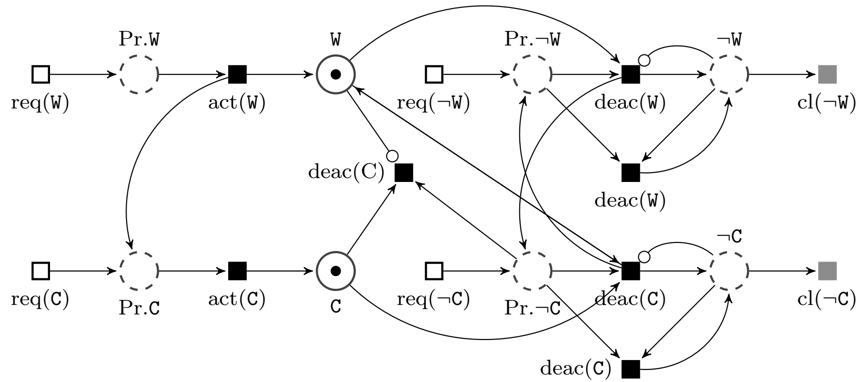

Fig. 4: Strong inclusion from `WiFi` (`W`) to `Connectivity` (`C`).

Requirement A requirement represents the situation in which activation of a context is possible only if another context is already active. This restriction implies that when the latter context is no longer active the former context must be deactivated. The CoPN corresponding to this interaction is shown in Fig. 5: `VideoCall` can be activated only if `HighBattery` is already active.

Thus far, contexts and dependency relations have been discussed as isolated CoPNs in the system. We now explain how different CoPNs can be composed to form a unified CoPN that the system can use as run-time model of the execution environment as a whole.

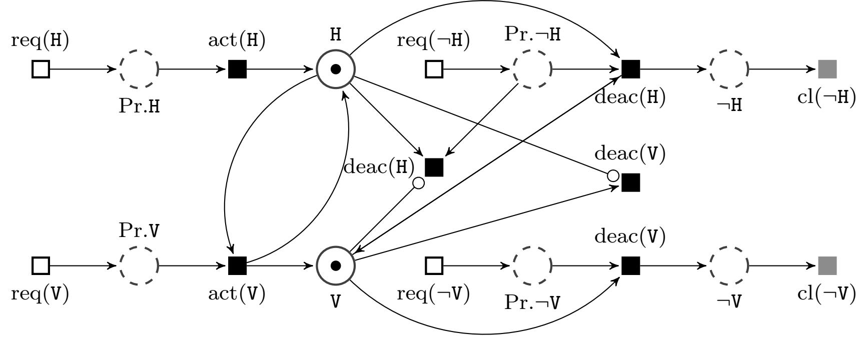

Fig. 5: Requirement of `HighBattery` (`H`) by `VideoCall` (`V`).

3.4 Composing Context Dependency Relations

This section provides an intuitive description of the steps needed to compose CoPNs.[5] A context only interacts with other contexts directly related to it. This provides modularity to the composition mechanism, because when composing, Petri net elements (arcs) are added only between the contexts being composed.

To preserve the semantics of dependency relations, CoPN composition extends the place combination technique of Petri nets [19]. As mentioned in the previous section, each dependency relation is comprised of a set of rules. Such rules must be verified to hold in the composed CoPN, after combining corresponding places and transitions. The verification process may add additional arcs when needed, to satisfy the rules.

Snippet 1 shows pseudo-code describing the composition of CoPNs. We explain the composition by means of an example of two dependency relations $R_1(C_1, C)$ and $R_2(C_2, C)$ between contexts C_1, C_2 and C. For simplicity, we assume that the two relations are to be composed into an empty CoPN $\mathcal{P}$. The first step in the composition is to combine the C context common to both relations. This is done by taking the union of all corresponding elements associated to each context –that is, elements with the same label; inputs and outputs are collapsed into one (lines 3–6). Second, for all existing dependency relations in the CoPN each rule is checked to ensure that it is satisfied. Additional arcs might be added for transitions that match a rule but do not satisfy it (lines 7–10).

```
1  add C1 to P
2  add C2 to P
3  loop for e1 such that e1 ∈ P_C ∪ T_C in R1
4           e2 such that e2 ∈ P_C ∪ T_C in R2
5     add e1 to P
6     if e1 ≠ e2 then add e2 to P
7  loop for R dependency relation in P
8           c constrain rule in R
9           t transition in P
10    if t does not satisfy c then add new arc(t,c) to P
```

Snippet 1: CoPN composition algorithm.

[5] A full formal description of composition in CoPNs falls outside the scope of this paper, but it is available as technical report [3].

3.5 Programming Support for Context Petri Nets

The CoPN model can become complex as the system grows. However, developers interact with it through a language abstraction layer that hides such complexity. This section presents the context-oriented constructs of Subjective-C, and how these map to the underlying CoPN model.

```
Context declaration ::= @context( context-name [,bound] )
Context activation ::= @activate( context-name )
Context deactivation ::= @deactivate( context-name )
Dependency relations declaration ::=
    [ addExclusionBetween: context-name and: context-name ]
    [ addWeakInclusionFrom: context-name to: context-name ]
    [ addStrongInclusionFrom: context-name to: context-name ]
    [ addRequirementTo: context-name of: context-name ]
```

Fig. 6: Subjective-C method syntax to interact with CoPNs.

Fig. 6 shows the language constructs available in Subjective-C for the creation and manipulation of contexts, and hence CoPNs. A *context declaration* automatically generates a context structure as that of Fig. 1. The maximum number of times a context can be activated can be *bounded* by a positive integer. Context *activation* and *deactivation* fire the corresponding external transitions in the underlying CoPN, for example req(`VideoCall`) and req(¬`VideoCall`) in Fig. 1. Finally, a *dependency relation declaration* specifies the different dependency relations between two contexts, as described in Section 3.3.

For illustration, Snippet 2 shows definitions for `LowBattery` and `HighBattery` contexts. Lines 1 and 2 generate a CoPN as that of Fig. 1 for each context. The exclusion dependency defined between the two contexts in line 3 yields the CoPN shown in Fig. 2. Line 4 is the activation of the `LowBattery` context which (when successful) installs the behavior adaptations associated to it. Due to the `LowBattery` context being active, activation of the `HighBattery` context in Line 5 is denied and the cause of the denial is given to the user.

```
1 SCContext *lb = @context(LowBattery);
2 SCContext *hb = @context(HighBattery);
3 [addExclusionBetween: lb and: hb];
4 @activate(LowBattery);
5 @activate(HighBattery);
```

Snippet 2: Example of exclusion dependency declaration.

4 Consistent Composition of Context-Dependent Behavior in CoPN

Having explained the core of the CoPN model in Section 3, we now turn to the question of how the model satisfies the requirements for consistent composition of context-dependent behavior put forward in Section 2.

4.1 Dynamic Context Activation and Deactivation (R.1)

CoPN provides a concrete representation of the system's context. The dynamic activation and deactivation of a context uses the definition of *consistent state* for a CoPN given in Section 3.2, which is ensured by the following process:

- Before external transitions are fired, the set of current active contexts is saved as the *current marking* of the system.
- If an inconsistency exists after firing all enabled internal transitions (that is, if a temporary place is still marked), *all* modifications made since the external transition firing are reverted. This is done by reinstating the previously saved current marking.
- In case an inconsistency exists, a message is prompt to the user with the reason preventing the activation or deactivation to take place.
- If the system reaches a consistent state, the current marking is updated to the marking found in the CoPN. A trace of all fired internal transitions is given to the user.

As an example, consider the discovery of a `Wifi` network connection in the mobile phone. The initial marking m_0 of the CoPN representing the `Wifi` context is m_0(`Wifi`)=0 which is a consistent state. When a `Wifi` network connection is discovered, this generates an `@activate(Wifi)` message. The transition to request the context activation is fired, *req*(`Wifi`), adding a token to the temporary place *Pr*.`Wifi`. This changes the initial marking m_0 to a new marking m_1, where m_1(*Pr*.`Wifi`) =1. Such a marking enables the internal transition *act*(`Wifi`) which now *must* fire according to the internal transition semantics described in Section 3.2. The firing moves the token from *Pr*.`Wifi` to `Wifi` yielding a marking m_2 where m_2(`Wifi`)=1. At this point none of the internal transitions is enabled, and none of the temporary places are marked. Therefore, the CoPN is in a *consistent state*. The case of context deactivation is similar to the context activation one.

4.2 Consistent Interactions Between Multiple Contexts (R.2)

The CoPN model ensures the consistent state of a system in presence of multiple active contexts by means of dependency relations and dynamic context activations. As explained in Section 3.3, CoPN currently supports the 4 dependency relations defined in Subjective-C. Dependency relations encode interactions between contexts. Such interactions define sequences of activations and deactivations that leave the system in a consistent state.

The activation or deactivation of a context is constrained by the existing dependency relations. For example, in the case of the requirement dependency relation of Fig. 5, the `VideoCall` context is only activated when the `HighBattery` context is already active. Were this not be the case, the activation of `VideoCall` would leave the system in an inconsistent state, and is therefore retracted.

4.3 Multiple Activations of the Same Context (R.3)

In CoPN, contexts can be activated multiple times. To support this behavior, CoPNs rely on the fact that a place can hold many tokens at once. Each token

represents an activation of the context. As such, the context (and thus the adaptations associated to this context) will remain available for as long as there are tokens in the context place. Fig. 7 shows an example where three internet protocols are available (e.g., `Edge`, `Wifi`, `3G`) for the mobile phone. This condition is represented in the `Connectivity` context by three tokens.

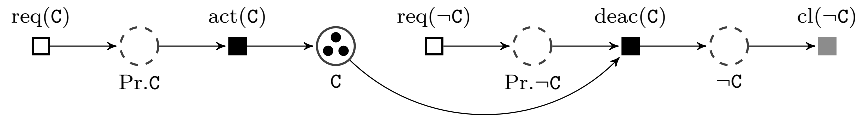

Fig. 7: Context `Connectivity` (C) is active three times.

Unlike existing COP approaches, CoPN allows developers to declare the activation of a context as *multiple* or *single*. For example, the generic `Connectivity` context can be activated multiple times, whereas a specific protocol like `3G` should be activated at most once. This extends existing COP approaches, which support either single-activation contexts (e.g. ContextL [4]), or multiple-activation contexts (e.g. Subjective-C), but not both.

5 Related Work

This section reviews related work by going through different context sensing approaches which provide a concrete representation of context, and by considering alternative modeling approaches that could be used for context-aware systems.

5.1 Context Representation

The Context Toolkit [20] and WildCat [5] frameworks provide abstractions for the representation of context information. Context information coming from sensors is represented by context objects. Gathered information can be contradictory or inconsistent. It is up to the system/developer to manually manage such inconsistencies.

CORTEX [21] is a middleware architecture that exploits the sentient object paradigm: so-called sentient objects receive events as input (from other sentient objects or sensors), process the events by means of an inference engine and generate further events as output. The sentient object model of CORTEX is intended for pro-active context-aware systems that autonomously invoke some action in response to relevant context changes. In contrast, our model deals with reactive systems. That is, upon a context change, the most appropriate context representation is activated.

These framework approaches provide useful modularization features to manage context information. However, they have little support for the dynamic activation of behavioral adaptations, and managing conflicts between them, making them ill-suited in face of the requirements presented in Section 2.

5.2 Alternative Approaches

CoPN serves both as a formal and run-time model of context. We now consider other approaches that could be used for the same purpose.

State Diagrams Automata [12] and statecharts [15] are used to describe system behavior based on its possible states, and the set of actions to be taken at each state. Automata and statecharts are normally used to verify system properties, such as program termination. Among the properties provided by these diagrams, composition is the most prominent. However, the system focuses only on one state at a time, this means that every state needs to associate all possible actions in the system. In the context of COP, where context activations represent the actions in the system, every state has to be connected to all such actions, making the model cluttered and complex. Additionally, both automata and statecharts formalisms would need to be extended to allow the interaction between contexts and multiple activations of a context.

Process Algebra, Coalgebra and Modal Logics Process algebra [11] is used to model concurrent processes, providing high-level abstractions for operations between processes such as parallel composition, communication, replication, and synchronization. Modal logics [17] have been used to represent necessity and possibility conditions about system properties. Modal logics are mostly used to express temporal conditions, but they also can be used to express conditions like program termination. Coalgebras [13] have been used to express dynamic behavior of systems. Typically, coalgebras specify state-based systems, where the state is considered as a black box and dynamic behavior is reasoned upon in terms of invariance and bisimilarity.

These formal methods could be used to model and reason about context-aware systems. However, concrete models based on these formalisms would need to be extended to match the requirements of Section 2, as we have done with the Petri net extensions used in CoPNs.

6 Future Work

Although the CoPN model can help in tackling some of the challenges for the consistent composition of behavioral adaptations, a number of challenging issues need to be further explored.

First, conflicts between external, internal and cleaning transitions are avoided by the separation of each class by their transition priorities. The question still remains, however, if within internal transitions conflicts exist. That is, if firing of a transition disables a previously enabled one, leading to different markings. Although, these type of conflicts are expected from the non-deterministic choice of transitions with the same priority, it should be proven that regardless of the firing order of transitions the same marking is always reached.

Second, CoPN provides consistency of dynamic behavior adaptations. However, the discussion presented in this work focuses on the management of interaction between contexts. How to identify such interactions, remains an open

question. Standard Petri net analysis techniques allow to reason about a system's behavior [16]. Such techniques could be used to identify interaction between contexts. The properties that could be used in the context of COP systems comprise (a) *reachability*, to identify if it is possible to have a particular configuration (i.e. marking) of active contexts, (b) *liveness*, to verify if a context can ever be activated or not, and (c) *persistency*, to spot isolated contexts in the Petri net. This analysis techniques can give upfront information about errors and redundancies in the system. Currently the CoPN model contains inhibitor arcs and is (in principle) unbounded, which makes these properties undecidable. However, the addition of bounds to contexts, and removal of inhibitor arcs when possible [18], could enable the analysis of such properties. We are currently studying which properties can be successfully verified for CoPN.

7 Conclusions

Ensuring consistent behavior adaptation of software systems is a challenging task. Inconsistencies in the composition of context-dependent behavior rise from interactions when such behavior is incompatible or contradictory. We identify three main requirements to support behavioral adaptations: dynamic context activation and deactivation, consistent interaction between multiple contexts, and multiple activations of the same context. As a way to address these requirements, this paper presents the context Petri nets (CoPN) model which builds on the dynamic activation and deactivation of contexts provided by context-oriented programming (COP) languages. CoPN uses different Petri net extensions to provide a precise and live representation of context, dependency relations between contexts, and their composition. The CoPN model makes explicit the different states in the activation life cycle of a context to cope with the reactive nature of COP systems, and to ensure that activations and deactivations are consistent. Consistent activation and deactivation of contexts is ensured by dynamically checking context dependency relations. If an inconsistency is encountered, CoPNs allows to rollback the faulty operation to the last registered consistent state. Afterwards, the user is informed of the cause of the error.

For the advantages provided in the management and assurance of consistent dynamic adaptations, context Petri nets are a convenient run-time representation of contexts, their activation and interaction in COP systems.

References

1. Bause, F.: On the analysis of petri nets with static priorities. In: Acta Informatica. vol. 33, pp. 669 – 685 (1996)
2. Best, E., Koutny, M.: Petri net semantics of priority systems. Theoretical Computer Science 96, 175–215 (April 1992)
3. Cardozo, N., González, S., Mens, K., D'Hondt, T.: Context petri nets: Definition and manipulation. Tech. rep., Université catholique de Louvain and Vrije Universiteit Brussel (April 2012), `http://soft.vub.ac.be/Publications/2012/vub-soft-tr-12-07.pdf`

4. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium. pp. 1–10. ACM Press (Oct 2005), co-located with OOPSLA'05
5. David, P.C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: Terzis, S., Donsez, D. (eds.) MPAC. ACM International Conference Proceeding Series, vol. 115, pp. 1–7. ACM (2005)
6. Desmet, B., Vanhaesebrouck, K., Vallejos, J., Costanza, P., Meuter, W.D.: The puzzle approach for designing context-enabled applications. In: XXVI International Conference of the Chilean Computer Science Society. pp. 23 – 29. IEEE (2007)
7. Eshuis, R., Dehnert, J.: Reactive petri nets for workflow modeling. In: Application and Theory of Petri Nets 2003. pp. 296–315. Springer (2003)
8. González, S., Cardozo, N., Mens, K., Cádiz, A., Libbrecht, J.C., Goffaux, J.: Subjective-C: Bringing context to mobile platform programming. In: Proceedings of the International Conference on Software Language Engineering. Lecture Notes in Computer Science, vol. 6563, pp. 246–265. Springer-Verlag (2011)
9. González, S.: Programming in Ambience: Gearing Up for Dynamic Adaptation to Context. Ph.D. thesis, Université catholique de Louvain (Oct 2008), `http://hdl.handle.net/2078.1/19684`, coll. EPL 211/2008. Promoted by Prof. Kim Mens
10. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: Proceedings of the Dynamic Languages Symposium. pp. 77–88. ACM Press, New York, NY, USA (Oct 2007), co-located with OOPSLA'07
11. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge, Mass. (1988)
12. Hopcroft, J.E., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
13. Jacobs, B.: Exercises in coalgebraic specification. In: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. pp. 237–280 (2000)
14. Kamina, T., Aotani, T., Masuhara, H.: Eventcj: A context-oriented programming language with declarative event-based context transition. In: Proceedings of the International Conference on Aspect-Oriented Software Development. pp. 253–264. AOSD'11, ACM Press (Mar 2011)
15. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of uml statechart diagrams. In: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). pp. 465–482. Kluwer, B.V., Deventer, The Netherlands (1999)
16. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541 – 580 (April 1989)
17. P. Blalckburn, M. de Rijke, Y.V.: Modal Logic. Cambridge University Press (2001)
18. Reinhardt, K.: Reachability in petri nets with inhibitor arcs. Electronic Notes in Theoretical Computer Science 223, 239–264 (2008)
19. Reisig, W.: Simple composition of nets. In: Proceedings of the 30th International conference on Applications and Theory of Petri Nets. pp. 23 – 42. Springer-Verlag (June 2009)
20. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: Aiding the development of context-enabled applications. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 434–441. ACM Press (1999)
21. Sørensen, C.F., Wu, M., Sivaharan, T., Blair, G.S., Okanda, P., Friday, A., Duran-Limon, H.: A context-aware middleware for applications in mobile ad hoc environments. In: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing. pp. 107–110. MPAC '04, ACM, New York, NY, USA (2004)