Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets

Nicolás Cardozo[1,2], Sebastián González[1], Kim Mens[1], Ragnhild Van Der Straeten[2], and Theo D'Hondt[2]
[1] ICTEAM Institute, Université catholique de Louvain. Louvain-la-Neuve, Belgium.
[2] Software Languages Lab, Vrije Universiteit Brussel. Brussels, Belgium.
e-mails: {nicolas.cardozo, s.gonzalez, kim.mens}@uclouvain.be, {rvdstrae, tjdhondt}@vub.ac.be

Abstract—**The development of self-adaptive systems requires the definition of the parts of the system that will be adapted, when such adaptations will take place, and how these adaptations will interact with each other. However, foreseeing all possible adaptations and their interactions is a difficult task, opening the possibility to inconsistencies or erroneous system behavior. To avoid inconsistencies, self-adaptive systems require a sound programming model that allows to reason about the correctness of the system in spite of its dynamic reconfigurations. This paper presents context Petri nets, a Petri net-based programming model for self-adaptive systems. This model provides a formal definition of adaptations and their interaction, as well as a consistent process for their inclusion in the system. Besides serving as an underlying run-time model to ensure that adaptations and their constraints remain consistent, context Petri nets allow to analyze properties such as reachability and liveness in the configuration of self-adaptive systems. Context Petri nets thus are a convenient tool to model and analyze the dynamics of self-adaptive systems, both formally and computationally.**

Index Terms—**Self-adaptive systems, Petri nets, dynamic behavior adaptation, context awareness.**

I. Introduction

Current-day computing devices have evolved from stand-alone computers to highly mobile systems with access to rich context information. Applications developed with context in mind can leverage the full potential of these systems by adapting their behavior dynamically according to sensed context changes. To support developing such applications, self-adaptive systems [9] emerged, allowing the definition, composition, and management of behavioral adaptations according to the system's surrounding context at run time.

Management and analysis of adaptations is key to the development of self-adaptive systems, to ensure that dynamic adaptations do not introduce errors at run time. To assure that adding or removing behavioral adaptations does not yield unexpected application behavior, most self-adaptive system architectures propose an external adaptation engine. Such engine is in charge of gathering information about the surrounding context, and reasoning about available adaptations and their appropriateness. Another alternative to implementing self-adaptive systems is the internal adaptation approach [11], which intertwines the application logic with the adaptation logic. Whereas the internal approach simplifies the composition and run-time infrastructure for interaction and management of adaptations, it does not provide any facilities to test or maintain the system and is not scalable. The external approach, on the other hand, is heavy-weight, requiring dedicated middleware for the implementation of self-adaptive systems. However, the external approach is a highly reusable approach, allowing the configuration of the engine for different systems.

This paper proposes a Petri net-based formalization and programming model to meet the tradeoff between the external and internal approaches for the implementation of self-adaptive systems. The proposed programming model is coupled with a programming language realizing the Context-Oriented Programming (COP) paradigm [12]. COP languages allow to dynamically add and remove behavioral adaptations at run time. Additionally, the formal basis of the proposed programming model enables us to separate the modeling and analysis of behavioral adaptations from the system's core logic.

We use our Petri net-based model for two main purposes. On the one hand, it provides a concrete semantics for the definition of adaptations and their interaction. On the other hand, it allows to analyze the dynamics of adaptations at design and run time, allowing to prove the coherence and consistency of adaptation definitions.

II. Dynamic Behavioral Adaptations

Self-adaptive systems tackle the problem of system reconfiguration and redeployment by the introduction of dynamic adaptations. In particular, the behavior of the system may be adapted whenever more appropriate behavior is available [9]. To enable adaptability, the system needs to be able to reason about itself and its surrounding context at run time. Self-adaptive systems, hence, propose an adaptation process shown in Fig. 1. The *context sensing* module gathers information about the system's surrounding context. The *analysis engine* module reasons about gathered information to select appropriate behavioral adaptations. The *context manager* module takes selected behavioral adaptations and composes them with the system's application logic. Finally, the *application behavior* module is the observable behavior of the system, which can have an effect on its context.

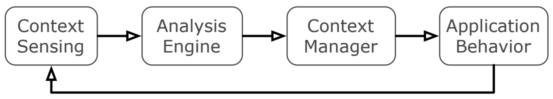

Fig. 1: Adaptation process for self-adaptive systems.

COP programming languages [12] have been specially engineered for dynamic behavioral adaptation, by allowing the addition and removal of behavior at run time. Such behavioral adaptations take place based on the sensed *context* of execution —that is, the reification of semantically meaningful situations in which the system executes [7]. Such situations can be endogenous (e.g. low battery charge, high CPU load) or exogenous (e.g. city location, weather conditions). With our notion of context, we put forward the circumstances under which a system executes as first-class entities that can be dealt with by the system. Whenever one of such situations is sensed, the corresponding context becomes *active*, and as a consequence the associated behavioral adaptations are dynamically deployed in the system [12]. When such particular situation no longer holds, the corresponding context becomes *inactive* and the associated behavioral adaptations are withdrawn from the system.

We use Subjective-C [7], a language extension of Objective-C, as representative example of a COP language. In Subjective-C, contexts are declared through the `@context(CPUHighLoad)` construct —in this case declaring a context representing that the system is running with high CPU load. The activation state of a context can be modified by means of two additional constructs, `@activate(CPUHighLoad)` and `@deactivate(CPUHighLoad)`. These constructs respectively add and remove behavioral adaptations associated with context `CPUHighLoad`. Finally, the behavioral adaptations for a particular context are defined by prepending a `@contexts` annotation to a regular method definition, as shown in Snippet 1.

```
@contexts CPUHighLoad
 - (Image *) decodeFrom:(Stream)stream {
     Image frame;
     // CPU-friendly decoding algorithm...
     // ... though of less quality
     return frame;
}
```

Snippet 1: Adaptation to `CPUHighLoad` context.

The language abstractions provided by COP languages, such as the one illustrated in Snippet 1, allow to modularize the system by clearly differentiating between base logic and adaptation logic. COP languages address the problems of reusability and scalability of internal approaches for the implementation of self-adaptive systems. However, to the best of our knowledge there are no COP languages that provide means to analyze and test behavioral adaptations for correctness. The analysis engine module illustrated in Fig. 1 is thus missing. In the following sections we show how we fill this gap.

III. Modeling Self-Adaptive Systems

To deal with the dynamics of behavioral adaptations in self-adaptive systems, we introduce a formal and run-time model called *context Petri nets (CoPNs)* [4]. The CoPN formalism provides a precise definition of contexts and their dynamic activation and deactivation. Additionally the formalism allows to define interaction between behavioral adaptations at run time. The verification of such interaction is explained in Section IV.

A. Context Petri Nets

This section formally defines CoPN and maps its elements onto COP concepts.[1]

Definition 1: A *context Petri net* is a 9-tuple $\mathcal{P}$=$<P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0>$ defined as a reactive Petri net [6] with inhibitor arcs [5], static priorities [1], and token colors [8], where P_c is the set of context places, P_t is the set of temporary places, T_e is the set of external transitions, T_i is the set of internal transitions, f is the flow function defining arcs between places and transitions, $f_\circ$ is the flow function defining inhibitor arcs from places to transitions, ρ is the function assigning transitions priorities, $\mathcal{L}$ is the set of token colors, and m_0 is the initial marking of the CoPN.

Places and transitions in a CoPN are given a label representing their intention. For example, a place labeled Pr(`A`) corresponds to context `A` preparing to activate, and a transition labeled act(`A`) corresponds to the activation of context `A`. Note that different transitions may have the same label. In such a case transitions are differentiated by their input ($\bullet t$ and $\circ t$) and output ($t\bullet$) places.

Definition 2: A CoPN corresponding to a *singleton context*, `A`, is defined as $C_{\texttt{A}}$=$<P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0>$, with context place $P_c = \{\texttt{A}\}$, temporary places $P_t = \{Pr(\texttt{A}), Pr(\neg\texttt{A})\}$, external transitions $T_e = \{req(\texttt{A}), req(\neg\texttt{A})\}$, internal transitions $T_i = \{act(\texttt{A}), deac(\texttt{A})\}$, the priority function ρ is given by the rules $\forall\, t_e \in T_e,\ \rho(t_e) = 0$, and $\forall\, t_i \in T_i,\ \rho(t_i) = 2$, $\mathcal{L} = \{black\}$, the flow function for inhibitor arcs has an empty domain, and f is defined as: f(req(`A`), Pr(`A`))=1, f(Pr(`A`), act(`A`))=1, f(act(`A`), `A`)=1, f(`A`, deac(`A`))=1, f(req(¬`A`), Pr(¬`A`))=1, f(Pr(¬`A`), deac(`A`))=1. In this particular case, the initial marking is $m_0(\texttt{A}) = 1$.

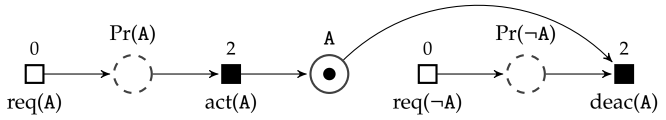

Fig. 2: CoPN $\mathcal{P}_{\texttt{A}}$ for context `A`.

Fig. 2 shows the visual representation of the singleton CoPN C_A. Hereafter, transition priorities are omitted from figures, as they can be deduced from the color of the transitions: the darker the color the higher the priority.

[1]An implementation of CoPN as the run-time model of Subjective-C is available at http://released.info.ucl.ac.be/Tools/Context-PetriNets.

Places in CoPNs capture the possible states of contexts:

- *Context places* (solid circles in Fig. 2) represent the context itself (e.g. context `CPUHighLoad`).
- *Temporary places* (dashed circles in Fig. 2) represent preparatory states for a context, easing the consistency verification and composition processes. The preparatory places Pr(A) or Pr(¬A) are used to process requests to respectively activate and deactivate a context A. Section IV-A illustrates the use of temporary places to manage consistent context activation.

Transitions represent actions that can be taken on the state of a system. In the case of CoPN, these actions correspond to context activations and deactivations. Activation and deactivation of contexts do not occur immediately, but need to be requested first and processed carefully, since the request may be denied if such action would violate constraints imposed by other contexts.

- *External transitions* (white squares in Fig. 2) are used to *request* a context activation or deactivation in response to changes in the surrounding environment.
- *Internal transitions* (black squares in Fig. 2) deal with interactions between contexts as discussed in Section III-C. Internal transitions trigger the actual activation or deactivation of contexts.

Tokens represent context activations. The state of a context is determined by the current marking of the CoPN. In Fig. 2, context A is *active* if the place labeled A contains a token; *prepared for activation* if place Pr(A) contains a token, and *prepared for deactivation* if place Pr(¬A) contains a token. The number of tokens in a context place represents the number of times the context has been activated.

Token colors represent the scope in which context activations take place. As a convention for CoPNs we use black for context activations with global scope (i.e. adaptations for the whole system), and we use other colors for local context activations (i.e. adaptations for one execution thread) [3]. There can be as many tokens as different local activations are requested.

Inhibitor arcs provide the possibility to verify the absence of tokens in a place. Inhibitors model interaction between contexts, for example to express that a context can be activated only if some other context is inactive. Inhibitor arcs are illustrated as circle-ended arcs ($\multimap$).

B. Dynamics of CoPNs

CoPNs are not static structures. On the contrary, CoPNs make it possible to represent and track the changes that occur in the surrounding environment while the system runs. CoPNs can thereby be used as run-time representations of contexts and their dynamic changes. The following descriptions explain the way context state is encoded in a CoPN, and how such state evolves according to encoded CoPN constraints, given $\mathcal{P}= <P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0>$.

- A transition t is *enabled* at a marking m for a color $l \in \mathcal{L}$, written $m[t\rangle_l$ if $\forall p_{in} \in \bullet t,\ f(p_{in}, t) \leq m_l(p_{in})$, $\forall p_{in} \in \circ t$, $m(p_{in}) = 0$, and $\nexists t'$ such that $\rho(t') > \rho(t)$ and $m[t'\rangle_l$
- Transition *firing* for a color $l \in \mathcal{L}$, leads from marking m to marking m', written $m[t\rangle_l m'$, where $\forall l \in \mathcal{L}$ and $\forall p \in P_c \cup P_t,\ m'(p) = m(p) - f(p,t) + f(t,p)$.
- 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 a consequence of a change in the surrounding environment.
- Internal transitions are fired with a *must* fire semantics. That is, if an internal transition is enabled it must fire. Whenever two internal transitions are enabled, they fire non-deterministically.
- A global activation holds also as local activation: black plays the role of any other possible color [3].

C. Context Dependency Relations in CoPN

In a COP system, contexts can depend on each other. That is, a context (de)activation can take place, or be refused, as a consequence of the (de)activation of other contexts. The interaction between contexts can be encoded by connecting transitions of a context with places of another one via (inhibitor) arcs. Such interactions constitute *context dependency relations* describing the activation and deactivation of a context with respect to other interacting contexts.

CoPN currently supports seven context dependency relations, these are: *implication* (−►), *requirement* (−◄), *exclusion* (□−□), *causality* (−▷), *suggestion* (- -▷), *conjunction* (→), and *disjunction* (−◇). Due to space limitations, we only show the definition of the implication and requirement dependency relations. Each context dependency relation is defined by a type and the set of singleton CoPNs between which the interaction is defined. Each context dependency relation is constructed by the application of two functions `ext` and `cons`, which respectively *extend* a set of singleton CoPNs and *constraint* the activation and deactivation of contexts. Other dependency relations are defined in a similar fashion.

*Definition 3 (**Implication**):* Implication dependency relations help encode, for example, containment between two physical contexts (e.g. `Brussels−►Belgium`), or provision of services (e.g. `WiFi−►Connectivity`).

Formally, the implication dependency relation (A−►B) between two singleton CoPNs C_A and C_B, is defined as the tuple $<I, C_A, C_B>$. The CoPN representing an implication dependency relation, $\mathcal{P}$=$<P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0>$ is obtained by the disjoint union of each of the singleton CoPNs ($\mathcal{P} = C_A \sqcup C_B$), and the application of the functions $\texttt{ext}_I$ and $\texttt{cons}_I$ over $\mathcal{P}$.

$\texttt{ext}_I$ is defined as $\texttt{ext}_I$:$(\mathcal{P}, < I, C_A, C_B >) \mapsto \mathcal{P}'$, such that $C_A, C_B \subset \mathcal{P}$ and $\mathcal{P}'$=$<P_c, P_t, T_e, T_i', f', f_\circ', \rho, \mathcal{L}, m_0>$, where $T_i' = T_i \cup \{\overline{deac}(\texttt{A})\}$, $f'(t,p) = f(t,p)$, and

$$f'(p,t) = \begin{cases} 1 & \text{if } p = \texttt{A} \wedge t = \overline{deac}(\texttt{A}) \\ f(p,t) & \text{otherwise} \end{cases}$$

$$f_\circ'(p,t) = \begin{cases} 1 & \text{if } p = \texttt{B} \wedge t = \overline{deac}(\texttt{A}) \\ 1 & \text{if } p = Pr(\texttt{B}) \wedge t = \overline{deac}(\texttt{A}) \\ f_\circ(p,t) & \text{otherwise} \end{cases}$$

The $\texttt{ext}_I$ function introduces a deactivation transition, $\overline{deac}(\texttt{A})$[2] for the source context of the dependency relation. This transition deactivates A whenever B is inactive and it is not preparing to activate.

$\texttt{cons}_I$ is defined as $\texttt{cons}_I$:$(\mathcal{P}, < I, C_A, C_B >) \mapsto \mathcal{P}'$, such that $C_A, C_B \subset \mathcal{P}$ and $\mathcal{P}'$=<$P_c, P_t, T_e, T_i, f', f_\circ, \rho, \mathcal{L}, m_0$>, where $f'(p,t) = f(p,t)$, and

$$f'(t,p) = \begin{cases} 1 & \text{if } \texttt{A} \in t\bullet \wedge \texttt{A} \notin \bullet t \wedge p = Pr(\texttt{B}) \\ 1 & \text{if } \texttt{A} \in t\bullet \wedge \texttt{B} \notin \circ t \wedge p = Pr(\neg\texttt{B}) \\ f(t,p) & \text{otherwise} \end{cases}$$

The arcs introduced by $\texttt{cons}_I$ respectively represent that: for every activation of A for which A is not an input, B is requested for activation, and for every deactivation of A for which B is not an inhibitor, B is requested for deactivation. Fig. 3 illustrates the CoPN representing the implication dependency relation $< I, C_A, C_B >$.

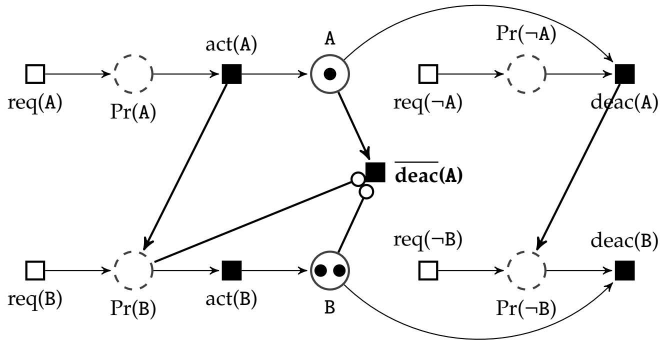

Fig. 3: Implication dependency relation (A–►B).

*Definition 4 (**Requirement**):* Requirement dependency relations are commonly used when one situation can occur only if another one is already taking place (e.g. HDVideo–◄LandscapeOrientation).

Formally, the requirement dependency relation (A–◄B) between two singleton CoPNs C_A and C_B, is defined as a tuple <Q, C_A, C_B>. The CoPN representing a requirement dependency relation, $\mathcal{P}$=<$P_c, P_t, T_e, T_i, f, f_\circ, \rho, \mathcal{L}, m_0$> is obtained by the disjoint union of each of the singleton CoPNs ($\mathcal{P} = C_A \sqcup C_B$), and the application of the functions $\texttt{ext}_Q$ and $\texttt{cons}_Q$ over $\mathcal{P}$.

$\texttt{ext}_Q$ is defined as $\texttt{ext}_Q$:$(\mathcal{P}, < Q, C_A, C_B >) \mapsto \mathcal{P}'$, such that $C_A, C_B \subset \mathcal{P}$ and $\mathcal{P}'$=<$P_c, P_t, T_e, T_i', f', f_\circ', \rho, \mathcal{L}, m_0$>, where $T_i' = T_i \cup \{\overline{deac}(\texttt{A})\}$, $f'(t,p) = f(t,p)$, and

[2]The $\overline{deac}(\texttt{A})$ label given to the transition is only used to easily identify it, however this label has no semantics.

$$f'(p,t) = \begin{cases} 1 & \text{if } p = \texttt{A} \wedge t = \overline{deac}(\texttt{A}) \\ f(p,t) & \text{otherwise} \end{cases}$$

$$f_\circ'(p,t) = \begin{cases} 1 & \text{if } p = \texttt{B} \wedge t = \overline{deac}(\texttt{A}) \\ f_\circ(p,t) & \text{otherwise} \end{cases}$$

The $\texttt{ext}_Q$ function introduces a deactivation transition, $\overline{deac}(\texttt{A})$, for the target context. This transition deactivates A whenever B is inactive.

$\texttt{cons}_Q$ is defined as $\texttt{cons}_Q$:$(\mathcal{P}, < Q, C_A, C_B >) \mapsto \mathcal{P}'$, such that $C_A, C_B \subset \mathcal{P}$ and $\mathcal{P}'$=<$P_c, P_t, T_e, T_i, f', f_\circ, \rho, \mathcal{L}, m_0$>, where

$$f'(p,t) = \begin{cases} 1 & \text{if } \texttt{A} \in t\bullet \wedge \texttt{A} \notin \bullet t \wedge p = \texttt{B} \\ f(p,t) & \text{otherwise} \end{cases}$$

$$f'(t,p) = \begin{cases} 1 & \text{if } \texttt{A} \in t\bullet \wedge \texttt{A} \notin \bullet t \wedge p = \texttt{B} \\ f(t,p) & \text{otherwise} \end{cases}$$

The arcs introduced by $\texttt{cons}_Q$ represent that, for every transition activating A, the transition is enabled if and only if B is active. Fig. 4 illustrates a CoPN representing the requirement dependency relation $< Q, C_A, C_B >$.

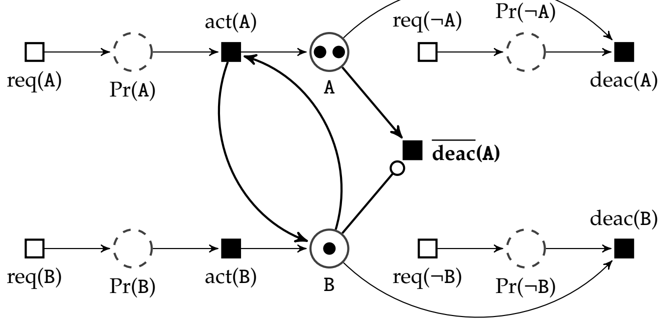

Fig. 4: Requirement dependency relation (A–◄B).

D. Composition of CoPNs

COP systems generally comprise multiple contexts. A CoPN is generated in three steps, given a definition of contexts and context dependency relations. First, all singleton CoPNs are united into a single CoPN. Second, the CoPN is extended according to all context dependency relations. Finally, the CoPN is constrained by adding arcs between contexts as required by all context dependency relations. The composition operator defined next generates a CoPN given a set of singleton CoPNs and a set of context dependency relations.

Definition 5: Lest S be a set of singleton CoPNs, and $\mathcal{R}$ be a set of context dependency relations. The composition of the contexts in S using the context dependency relations in $\mathcal{R}$ is defined as a CoPN $\mathcal{P}$=$\circ(S,\mathcal{R})$=$\texttt{cons}(\texttt{ext}(\texttt{union}(S),\mathcal{R}),\mathcal{R})$, where $\texttt{union}(S)$ is the disjoint union of all members of S into a CoPN $\mathcal{P}''$,

$\texttt{ext}(\mathcal{P}'',\mathcal{R}) \mapsto \mathcal{P}'$ is the application of $\texttt{ext}_R$, $\forall R \in \mathcal{R}$, and $\texttt{cons}(\mathcal{P}',\mathcal{R}) \mapsto \mathcal{P}$ is the application of $\texttt{cons}_R$, $\forall R \in \mathcal{R}$.

The constructive process by which CoPNs are composed ensures the satisfiability of all constraints imposed by all context dependency relations defined in the set $\mathcal{R}$.

Example 1: Let S={A,B,C} and $\mathcal{R}$={<I,A,B>, <Q,A,C>}. The composed CoPN, $\mathcal{P} = \circ(S,\mathcal{R})$, is shown in Fig. 5.

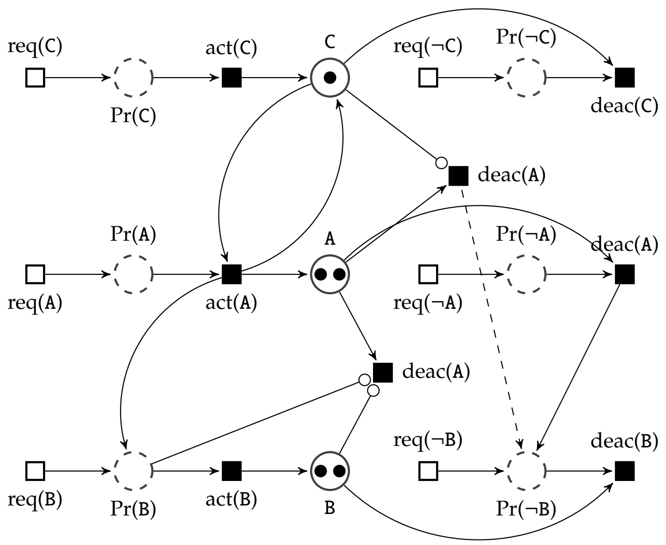

Fig. 5: Composed CoPN $\mathcal{P} = \circ(S,\mathcal{R})$.

Note that in the constraining function of the implication dependency relation, $\texttt{cons}_I$, takes into account all transitions defined in the CoPN —from all singleton CoPNs and those added by the two context dependency relations. As a result, the dashed arc ($deac$(A), Pr(¬B)) is added in Fig. 5 following the second case of the flow function extension of $\texttt{cons}_I$.[3]

Finally, note that in CoPN contexts and context dependency relations are not explicitly encoded by developers, rather they are defined through a dedicated DSL [7, 4]. This definition automatically composes the CoPN, following the process described in Definition 5.

IV. Analysis of Context Petri Nets

Having defined a formal model for contexts in self-adaptive systems, this section explores the analysis capabilities of such model. CoPNs offer two kinds of analyses. One analysis takes place at run time, and verifies the consistency of context activations with respect to the firing rules of CoPNs and the conditions imposed by context dependency relations. The second analysis takes place at design time, and reasons about the structural coherence of the CoPN and its static properties.

A. Run-Time Consistency Verification

Before explaining the process by which consistency of context activations is verified, we explain first what does it mean for a CoPN to be consistent.

[3]Here the arc is dashed to easily identify it, but this convention does not have a special semantic meaning.

Definition 6: Let m a reachable marking of a CoPN $\mathcal{P}$. A *step* Υ, from m is defined as a finite sequence of transitions $t_0, t_1, \ldots, t_n$ where $t_0 \in T_e$ and $t_1, \ldots, t_n \in T_i$, such that $m[t_0\rangle m_1[t_1\rangle \ldots m_n[t_n\rangle m'$. We say that m' is reachable from m via step Υ, and write it as $m[\Upsilon\rangle m'$.

Definition 7: A CoPN $\mathcal{P}$ is said to be in a *consistent state* for a marking m iff $\forall p \in P_t,\ m(p) = 0$.

Definition 8: Let $\mathcal{P}$ be a consistent CoPN, then Υ is a *consistent step* of $\mathcal{P}$ for a marking m, iff $m[\Upsilon\rangle m'$, and m' is a consistent state.

The activation process for a consistent CoPN is as follows: whenever a request to activate or deactivate a context A is triggered in the system, the corresponding external transition req(A) or req(¬A) is fired in the CoPN, and no other external transition is fired until there are no more internal transitions to be fired.

Definition 9: We define the *state* of a COP system as a triplet $< \mathcal{P}, \Sigma, \tilde{m} >$ given by its CoPN $\mathcal{P}$ with current marking m, where $\Sigma \subseteq T_i$ is a set of transitions to be fired, and $\tilde{m}$ the last consistent state of $\mathcal{P}$.

To simplify the notation, the rules introduce two auxiliary predicates, $\texttt{enabled}_m(T) = \{\, t \in T \,|\, t \text{ is enabled}\,\}$ and $\texttt{marked}_m(P) = \{\, p \in P \,|\, m(p) > 0\,\}$.

Rule 1: External transition firing occurs only at the beginning of a step Υ when $\mathcal{P}$ is in a consistent state $\tilde{m}$[4] and Σ is empty. After firing an external transition, the marking of the CoPN is modified, possibly enabling transitions in T_i. Such transitions become the elements in the set Σ.

$$(1)\ \frac{m = \tilde{m},\ \Sigma = \phi,\ t \in T_e,\ m[t\rangle m'}{< \mathcal{P}, \phi, \tilde{m} > \rightarrow < [m/m']\mathcal{P}, \texttt{enabled}_{m'}(T_i), \tilde{m} >}$$

Rule 2: Internal transition firing. If the set Σ is not empty, firing one of the internal transitions with highest priority yields a new marking m' of the CoPN. The new marking possibly enables some transitions in T_i, which become the elements of Σ.

$$(2)\ \frac{t \in \Sigma,\ m[t\rangle m'}{< \mathcal{P}, \Sigma, \tilde{m} > \rightarrow < [m/m']\mathcal{P}, \texttt{enabled}_{m'}(T_i), \tilde{m} >}$$

Rule 3: Evaluation termination. When there are no more internal transitions in the priority set Σ, two cases are possible. In the first case, if there are marked temporary places in the CoPN, all changes to the CoPN are rolled back by reverting to the last consistent state $\tilde{m}$:

$$(3.1)\ \frac{\Sigma = \phi,\ \texttt{marked}_m(P_t) \neq \phi}{< \mathcal{P}, \phi, \tilde{m} > \rightarrow < [m/\tilde{m}]\mathcal{P}, \phi, \tilde{m} >}$$

In the second case, if no temporary places are marked, we finish the step by turning the current marking m of the CoPN into the new consistent state $\tilde{m}$:

$$(3.2)\ \frac{\Sigma = \phi,\ \texttt{marked}_m(P_t) = \phi}{< \mathcal{P}, \phi, \tilde{m} > \rightarrow < \mathcal{P}, \phi, m >}$$

[4]Its current marking m equals the last consistent state $\tilde{m}$.

Theorem 1: Let $\mathcal{P}$ be a consistent CoPN. Every terminating step Υ, triggered by a request to activate or deactivate a context $\mathtt{A} \subseteq \mathcal{P}$, is a consistent step.

Proof: After firing an external transition, without loss of generality req(A), reduction rule (1) leads to the marking of at least one temporary place Pr(A). There are two cases for marking of temporary places:
1) If no internal transition is enabled after the external transition firing (i.e. if $\Sigma = \phi$), reduction rule (3.1) is applied, which rolls back $\mathcal{P}$ to its original consistent state. Thus, Υ is a consistent step.
2) If on the contrary, there are transitions in T_i to be fired, reduction rule (2) is applied as many times as needed. Every time, reduction rule (2) modifies the marking of $\mathcal{P}$ to a new marking m'. When eventually the set Σ becomes empty, one of the two Reduction rules (3.1) or (3.2) can by applied: rule (3.1) is applied when there remains a marked temporary place. The CoPN is then reverted back to its original consistent state, making Υ a consistent step; rule (3.2) is applied when no temporary place remains marked. Reduction rule (3.2) then updates to the new marking m' which is consistent because no temporary places are marked. Υ is a consistent step. ■

If a step Υ leads to an inconsistent state —that is, it leads to reduction rule (3.1)— then the step is oblivious to the system and the CoPN is reverted to the last consistent state. Whenever a context activation or deactivation is disregarded, this anomaly is reported back to the user providing the reason why the context (de)activation did not take place, for example "context A cannot be activated because context A is preparing to activate and cannot complete the operation (context B is inactive)."

Example 2: To demonstrate the dynamics of context activation and deactivation in CoPN, consider the sequence of commands σ = (`@activate(C)`, `@activate(A)`, `@activate(A)`, `@deactivate(C)`) for the CoPN c shown in Fig. 5 with an empty initial marking, $m_0(p) = 0\ \forall p \in P$.

After `@activate(C)`, `@activate(A)` and `@activate(A)` have been processed completely, the CoPN reaches a consistent state m, where $m(\mathtt{C})$=1, $m(\mathtt{A})$=2 and $m(\mathtt{B})$=2, as illustrated in Fig. 5. Execution of the `@deactivate(C)` command, triggers the firing of external transition $req(\neg\mathtt{C})$ yielding a new marking m_1, where $m_1(Pr(\neg\mathtt{C}))$=1, $m_1(\mathtt{C})$=1, $m_1(\mathtt{B})$=2, and $m_1(\mathtt{A})$=2. For this marking the only enabled internal transition is the deactivation transition $deac(\mathtt{C})$. Firing this transition (since it must happen) yields a marking m_2, where $m_2(\mathtt{C})$=0, $m_2(\mathtt{B})$=2, $m_2(\mathtt{A})$=2, and $m_2(Pr(\neg\mathtt{C}))$=0. Here, transition $deac(\mathtt{A})$ (between contexts A and C) becomes enabled because place C is no longer marked. Firing this transition yields a marking m_3, where $m_3(\mathtt{A})$=1, $m_3(\mathtt{B})$=2, and $m_3(Pr(\neg\mathtt{B}))$=1. At this point (the same) transition $deac(\mathtt{A})$ is enabled, and $deac(\mathtt{B})$ becomes enabled. Since the two transitions have the same priority, they can fire at random. Suppose that $deac(\mathtt{B})$ fires first. This leads to a marking m_4, where $m_4(\mathtt{A})$=1, $m_4(\mathtt{B})$=1, and $m_4(Pr(\neg\mathtt{B}))$=0. This marking does not enable any new transition. However, transition $deac(\mathtt{A})$ is in Σ, and must thus fire. The firing yields marking $m_5(\mathtt{B})$=1, and $m_5(Pr(\neg\mathtt{B}))$=1, enabling transition $deac(\mathtt{B})$. Firing the transition yields an empty marking, which is consistent.

B. Design-Time Coherence Analysis

To provide a more comprehensive analysis of self-adaptive systems, we extended the consistency verification of context activations with a reasoning engine to analyze incoherences that may be introduced while defining context dependency relations. So far CoPNs are ensured to always be consistent. However, the definition of context dependency relations may not always be coherent, in the sense that contexts may not be reachable.

*Definition 10 (**Coherence**):* A coherent CoPN is such that all context places are reachable, and there are no infinite steps.

Petri nets provide a set of properties based on the dynamics of transition firing. These properties can be used to identify conflicts in CoPNs. The most interesting properties to analyze about CoPNs are reachability and liveness. Given an initial marking, reachability verifies whether certain markings could ever occur in a Petri net. In the context of CoPNs, such analysis could be used to identify if a particular configuration of active contexts is possible, given an initial system state. Liveness (in its stronger version) means that no matter the marking of a Petri net, it is always possible to eventually fire all of its transitions. In the context of CoPNs, this could be used to verify if context activations (internal transition firings) can ever take place.

CoPNs pose a challenge when it comes to analyzing these properties. In the general case, Petri net analyses are undecidable in the presence of inhibitor arcs. However, there are conditions under which it is possible to analyze Petri nets with inhibitor arcs, in particular when there is only one, and also for primitive systems [10, 2].

To analyze CoPNs we unfold them to regular Petri nets. The unfolding is based on bounding the CoPN and stripping it down of its reactive and priorities semantics. Removing the reactive and priorities semantics is not a problem, given that these Petri net extensions can be unfolded to regular Petri nets without affecting their semantics [6, 1]. Unfortunately, giving a bound to the CoPN changes the initial semantics. Contexts are disallowed from activating beyond the bound, so transition firing sequences may become invalid (they do not reach a consistent state) because internal transitions may be disabled due to one of their outputs having reached its capacity. The unfolding of CoPNs is based in the unfolding of primitive systems [2]: (1) Every inhibiting place p with capacity N ($p \in \circ t$ for some transition t) is replaced by a set of places $\{p^i | i = 0, \ldots, N\}$; having a token in place p^i represents place p having i

tokens. (2) Inhibitor arcs to the transition t such that the inhibiting place $p \notin t\bullet$ are substituted by the arcs (p^0, t) and (t, p^0). (3) Each transition t incident to p is replaced by a set of transitions, each of which manages a specific representation of the contents of place p by means of places p^i: (*a*) if $p \in t\bullet$ then t becomes the set $\{t^i | i = 0, \ldots, N-1\}$; when t^i fires, it removes a token from p^i and adds a token to place p^{i+1}; (*b*) if $p \in \bullet t$, then t becomes the set $\{t^i | i = 1, \ldots, N\}$; when t^i fires, a token is removed from p^i and added to p^{i-1}.

From the transformation algorithm, we know that every accepted sequence of consistent steps in the unfolded Petri net is also an accepted sequence in the initial CoPN because the unfolding restricts the number of tokens in each place. This can be used, for example, to find the incoherence in the CoPN of Fig. 6.

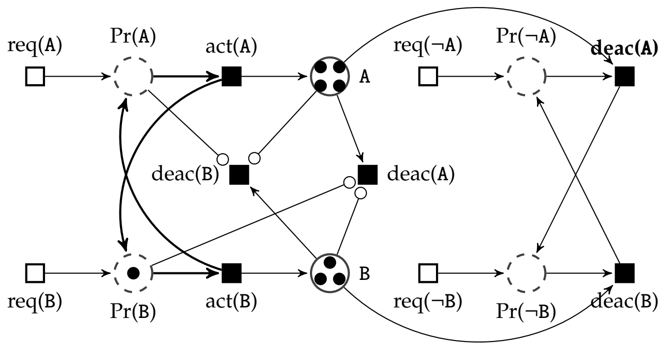

Fig. 6: Incoherent CoPN composition A–►B ∘ B–►A.

Once the CoPN has been unfolded into a bounded CoPN without inhibitor arcs, it can be analyzed by existing Petri net analysis tools such as LoLA [13]. LoLA is able to reason about reachability and liveness properties for the complete net, or particular markings, places or transitions.

Based on the context dependency relations defined in the CoPN, it is possible to analyze if the behavioral adaptations associated to each context will be available or not. Testing Petri net properties with LoLA requires the representation of the Petri net and the target state/element to be analyzed. Reachability analyses test final states of the CoPN; for example, if there is an state in which places A and B would be marked, and no other place would be in the CoPN of Fig. 6. Liveness analyses test if there are reachable states in which a particular transition is no longer fireable; for example, to test if the rightmost transition labeled deac(A) is dead.

In addition to the response to whether a property test is successful or not, LoLA can also provide outputs containing the state reached by the Petri net, and the firing sequence that led to such state. This information can be used by the developer to further assess validity with respect to the firing semantics of CoPNs (Section III-B).

V. Validation

As validation of our approach we developed a *Mobile City Guide* application using Subjective-C and CoPNs. The *Mobile City Guide* enables tourists visiting a city to navigate through the city's Points of Interest (POIs). The focus of this case study is on CoPN's compositionality, and analysis of system properties.

The application provides the possibility to create, customize and follow city tours based on a selection of POIs. The application consists of three main features:

F.I. *Tour Creation & Selection:* This feature allows users to create and select city tours based on a list of available POIs. A tour can be followed in two modes: if the device has a GPS, there is a GuidedTour mode which guides users throughout the city according to a predefined route of POIs; in FreeWalk mode, users walk freely around the city and can see directions to the POIs of their interest.

F.II. *City Navigation:* The application provides map or compass navigation to hop between POIs in the city, which can for example be used in FreeWalk mode when no GPS is available.

F.III. *POIs Display & Information:* POIs and their information can be displayed according to user-defined preferences such as UserLanguage, TargetAudience, or UserInterests.

These features can be enhanced further according to the environment in which the application executes.

1) A Language context allows to adapt displayed information to a particular language, such as English or French, where French–►Language and English□–□French. The language is either determined by geographical position (e.g. UK or France), for which Language–◄Positioning, or can be selected by the user in the application preferences via UserLanguage, where UserLanguage□–□Language. These contexts target the behavior of feature *F.III*.

2) A Connectivity context allows to fetch additional information about POIs whenever an internet connection is available: WiFi–►Connectivity. This can be used to stream POI descriptions as a VideoStream or AudioStream, (Connectivity–►AudioStream). Since video streaming is power consuming, VideoStream is only available if the device has a HighBattery level, hence VideoStream–◄HighBattery. These contexts target the behavior of features *F.I* and *F.II*.

3) A TimeOfDay context is used to display images associated with POIs. Depending on whether it is Day or Night (Day–►TimeOfDay), either images taken during the day or during the night are displayed. The TimeOfDay context also modifies the order in which POIs are displayed, according to their visiting hours. These contexts target the behavior of features *F.I* and *F.III*.

The initial model of the *Mobile City Guide* consisted of 20 contexts and 12 context dependency relations. The LoLA test cases expressing the conditions imposed by these context dependency relations are automatically generated, resulting in 57 test cases. Analyzing the results obtained through LoLA, it was possible to identify incoherences in the CoPN definition, for example between the Language–UserLanguage interaction with respect to the behavior of the VideoStream adaptation. After iterating over the adaptation model to address the identified problems, the adaptation model currently consists of 32 contexts and 34 context dependency relations.

VI. Related Work

ASSL [14] is a formal tool for modeling embedded systems with adaptive characteristics. ASSL allows to specify events coming from the hardware as self-management policies. However, ASSL does not provide a means to reason about the system. Zhang and Cheng [16] propose a model-based approach that separates the adaptive behavior from the regular behavior. This approach introduces state transfers, and global invariants that can be verified at run time by means of model checking. FORMS [15] is a high-level formal model specification proposed for the definition of self-adaptive systems. FORMS enables to describe and reason about architectural characteristics of self-adaptive systems.

The work presented in this paper differs from these approaches in that it takes a programming language perspective in contrast of an architectural approach. Similar to FORMS, our definitions allow to formally describe the adaptation dynamics of the system and interaction between adaptations, to reason about the composition of adaptations, and additionally to verify adaptation consistency at run time.

VII. Conclusion

The management of self-adaptive systems has proven a challenging task. Composition of adaptations may lead to unexpected or contradictory behavior if not dealt with carefully. Although the study of these problems is still incipient, a few modeling techniques have already been proposed. Unfortunately, these techniques are not entirely satisfactory, as many lack means to reason about the system. Those which support analyses are restrictive in the kind of adaptations they allow.

We propose the CoPN model as a formal and run-time model that enables the definition and analysis of self-adaptive systems. CoPNs allow the expression of adaptation dynamics, and closes the gap between specification and implementation of adaptations, allowing to ensure that context activations do not lead to inconsistent system states at run time. CoPNs ensure that composition of contexts will not break the constraints imposed by context dependency relations in the running system.

From our experience so far, context Petri nets have proved suitable both as a formal tool and as run-time representation of context in self-adaptive systems.

References

[1] F. Bause. *On the Analysis of Petri Nets with Static Priorities*. Acta Informatica. 1996.

[2] N. Busi. *Analysis of Petri Nets With Inhibitor Arcs*. Theoretical Computer Science 275 (2002).

[3] N. Cardozo, S. González, and K. Mens. *Uniting Global and Local Context Behavior with Context Petri Nets*. Intl. Workshop on Context-Oriented Programming. 3. ACM Press, 2012.

[4] N. Cardozo, J. Vallejos, S. González, K. Mens, and T. D'Hondt. *Context Petri Nets: Enabling Consistent Composition of Context-Dependent Behavior*. Intl. Workshop on Petri Nets and Software Engineering. CEUR-WS.org, 2012.

[5] G. Chiola, S. Donatelli, and G. Franceschinis. *Priorities, Inhibitor Arcs, and Concurrency in P/T Nets*. Intl. Conf. on Application and Theory of Petri Nets. 1991.

[6] R. Eshuis and J. Dehnert. *Reactive Petri Nets for Workflow Modeling*. Application and Theory of Petri Nets 2003. Springer, 2003.

[7] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. *Subjective-C: Bringing Context to Mobile Platform Programming*. Intl. Conf. on Software Language Engineering. Springer-Verlag, 2011.

[8] K. Jensen. *An Introduction to the Theoretical Aspects of Coloured Petri Nets*. A Decade of Concurrency, LNCS 803 (1994).

[9] R. Laddaga. *Self-adaptive software*. Tech. rep. 98-12. DARPA BAA, 1997.

[10] K. Reinhardt. *Reachability in Petri Nets with Inhibitor Arcs*. Electronic Notes in Theoretical Computer Science 223 (2008).

[11] M. Salehie and L. Tahvildari. *Self-adaptive software: Landscape and research challenges*. ACM TAAS 4.2 (2009).

[12] G. Salvaneschi. *Context-oriented Programming: A Software Engineering Perspective*. Journal of Systems and Software (2012).

[13] K. Schmidt. *LoLA: a low level analyser*. Intl. Conf. on Application and theory of Petri nets. Springer-Verlag, 2000.

[14] E. Vassev and M. Hinchey. *The ASSL approach to specifying self-managing embedded systems*. Concurr. Comput. : Pract. Exper. 24.16 (2012).

[15] D. Weyns, S. Malek, and J. Andersson. *FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems*. ACM TAAS 7.1 (2012).

[16] J. Zhang and B. H. C. Cheng. *Model-based development of dynamically adaptive software*. Intl. Conf. on Software engineering. ACM, 2006.