Distributed Context Petri Nets

Jose Daniel Fandiño de la Hoz, Juan Sebastián Sosa, Nicolás Cardozo
Systems and Computing Engineering Department Universidad de los Andes, Colombia
[jd.fandino10,js.sosa10,n.cardozo]@uniandes.edu.co

ABSTRACT

Dynamically adaptive software systems are inherently distributed. These systems enable the dynamic adaptation of systems' behavior according to information gathered from their environment, through sensors. As the system behavior is associated with different sensing devices, the complete system is not contained in a single device, but the complete system behavior can come from cooperation and communication between multiple nodes, as the environment evolves. However, existing context-oriented programming languages underlying such systems work under a closed world assumption, in which the system is completely contained and managed in a single node. This assumption hinders the applicability of dynamic adaptations in environments as those proposed in cyber physical systems. To develop the full potential of dynamically adaptive systems, we propose a run-time execution model, distributed context Petri nets, to manage the interactions of behavioral adaptations between remote nodes. Each node consists of a context Petri net extended with the capability to remotely interact with context Petri nets in other nodes. To validate the ability of our model to manage behavioral adaptations in distributed settings, we demonstrate the semantics of the different context dependency relations in face of different situations yielding inconsistencies.

CCS CONCEPTS

• **Software and its engineering → Distributed systems organizing principles**; *Semantics*; Context specific languages;

KEYWORDS

Distributed systems, Dependency relations, Consistency

ACM Reference format:
Jose Daniel Fandiño de la Hoz, Juan Sebastián Sosa, Nicolás Cardozo. 2019. Distributed Context Petri Nets. In *Proceedings of Workshop on Context-oriented Programming, London, United Kingdom, July 15, 2019 (COP'19),* 8 pages.

1 INTRODUCTION

The use of dynamically adaptive software systems [10] has seen a revival with the advent of Cyber Physical Systems (CPSs) [1]. CPSs are characterized by the interconnection of multiple sensor and actuator devices, with software services that utilize the information sensors provide to offer the most appropriate behavior, with respect to their environment, effected by actuators. CPSs are inherently distributed systems, as devices and services can be deployed in different physical machines and operate at different moments in time. Moreover, CPSs evolve over time, adding and removing devices and services, which in turn requires their observable behavior to change accordingly; enabling the possibility to build highly personalized systems through software adaptations.

The Context-oriented Programming (COP) paradigm [12] offers a modularization technique to realize adaptive systems by means of the dynamic (re)composition of software behavior at a fine granularity level. COP proposes language-level abstractions for the definition and manipulation of adaptations as behavior specializations for particular situations sensed from the systems' surrounding execution environment. In this paradigm, *contexts* are first-class entities that represent semantically meaningful situations from the system's surrounding execution environment. Contexts, are associated to *behavioral adaptations* —that is, fine grained behavior definitions (*e.g.,* objects, methods, object properties) to be applicable in specific situations. Dynamic adaptation takes place from the dynamic (re)composition of behavior, in response to *context activation* or *context deactivation* events. Whenever a context is activated or deactivated, in response to stimuli form the environment, all the behavioral adaptations associated to such context are composed with the system's executing behavior, effectively observing a dynamic system behavior.

As context activation and deactivation happens unannounced, and behavioral adaptations are not aware of each others behavior beforehand, the system behavior can reach inconsistent states. The Context Petri nets (CoPNs) run-time execution model is proposed to manage context inconsistencies [4] (Section 2). However, the CoPN model does not take into account the effect of nodes' unavailability, and therefore the adaptations defined therein, in a distributed setting. Such a close world vision prevents the applicability of COP for the development of CPSs in two perspectives. First, a close world vision, in which the system resides in a single node, is in contradiction with the essence of CPSs. Second, even if communication between nodes is possible, no existing COP model validates the consistency of the system in a distributed setting.

The objective of this paper is to manage contexts' consistency whenever they are defined across different nodes in a distributed system. To this effect, we propose an extension of CoPNs, called Distributed Context Petri nets (DCoPNs), that extends the semantics and management of context interaction, to assure consistent context interaction in distributed settings (Section 3). DCoPNs extend the semantics of CoPN by managing conflicts that arise from the connection and disconnections of CoPNs deployed in different nodes by means of a consensus algorithm. To validate the effectiveness of the proposed inconsistency detection and resolution, we

ACM ISBN …$15.00
DOI: 10.1145/3340671.3343359

explore all situations in which context activation and deactivation in different nodes interact as their nodes connect and disconnect with remote nodes (Section 4).

To put the problem and its solution in perspective we use the mobile city guide application [6, 7]. In this case study, users roam around a city following a visit guide. The guide organizes a sequence of Points of Interest (POIs) for users to follow. Along the visit information about the different POIs is displayed to users. The system configuration and displayed information is dynamic based on users' preferences and surrounding execution environment (*i.e.,* sensor provided information). As a consequence, specific behavior offered by the application depends on the capabilities currently deployed in the environment –that is, available sensors in users' vicinity. Given that the application is of general use in different cities, it is not possible to know all available devices beforehand. Therefore, the context specific to an environment is only known dynamically.

The mobile city guide application displays information, an image, and temporarily download additional available information or features about the current POI in the itinerary. The information to display is personalized to users (*e.g.,* `AgeGroup`). Notwithstanding, to avoid draining users devices' battery, the application does not fetch additional information if the system has a `LowBattery`. Additionally, specific information and features are available for each specific POI's sensor environment. For example, POIs offering internet access to their users (*e.g.,* `WifiConnectivity`) can present video explanations of the POI. Additionally, specific POIs could extend the user experience by tailoring the behavior according to user preferences (*e.g.,* `AgeGroup`). Again each of these contexts are available if the device providing the information is in users' proximity. Some of the sensor information moves along with the user (*e.g.,* GPS, device, user preferences). Other information is only applicable to certain places and their capabilities. Therefore, certain adaptations only make sense for those places, unknown to other parts of the application (*e.g.,* user's devices).

2 CONTEXT PETRI NETS

CoPNs [4] are defined as a run-time execution model for COP languages. The objective of CoPNs is to verify the consistency of context activation and deactivations as these take place in response to situations from the surrounding execution environment.

In CoPNs, each context is represented as a specific Petri net structure, similar to the one shown in Figure 1 for context `AG` (`AgeGroup`). This structure manages activation and deactivation of the context in response to stimuli from the surrounding environment. In CoPNs, contexts' activation state is represented by Petri net places. Each context can be in one of four different states, preparing to activate, active, preparing to deactivate, and inactive. The preparation states correspond to *temporary places* (dashed line places in Figure 1), which are transitional states of the context while its activation or deactivation are verified. The active state of a context corresponds to *context places*, marked once the context has been activated. Petri net tokens represent the current state in which the context is. If the context is inactive, then it has an empty marking. Petri net transitions represent the possible actions on a context. The white transitions, *external transitions*, respond to actions triggered from the environment —that is, calls to the `activate` and `deactivate` context abstractions. The black transitions, *internal transitions*, correspond to reactive transitions —that is, transitions that must fire immediately upon being enabled.

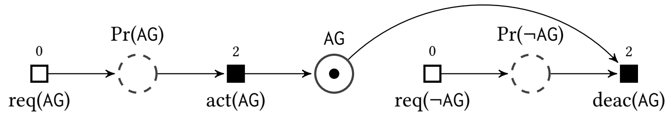

Figure 1: Singleton CoPN structure for the A context

Context activation, as well as the interaction between contexts, is managed by means of the Petri net token game semantics. A request to activate a context, using `activate(AG)`, triggers the firing of the `req(AG)` external transition (the white leftmost transition in Figure 1). This transition causes a token to generate in the `Pr(AG)` prepare to activate temporary place. If no conflict exits for the activation of the context, verifying the connection of this place and its associated internal transition with other contexts defined in the environment, then the activation is processed by the triggering of the internal transition, moving the token from `Pr(AG)` to the `AG` context place, which signals the context is now active. Context deactivation follows a similar procedure, taking into account the two transitions and temporary place right of the context place.

Interaction between contexts is defined as *context dependency relations* [5, 3]. CoPNs originally define seven context dependency relations between contexts, allowing different interactions when activating or deactivating contexts; these are: (1) exclusion, (2) causality, (3) implication, (4) requirement, (5) suggestion, (6) conjunction, and (7) disjunction. Multiple contexts can be composed in a single application with many different context dependency relations between them. Figure 2 show the example of a causality dependency relation between contexts `A` and `B`. The causality dependency relation, for example, represents a situation in which activation of context `A` triggers the activation of context `B`. Note that the transitions in the CoPN manage the interaction between the contexts. In Figure 2, for example, the transition in bold states the deactivation of context `A`, whenever context `B` is inactive, verified using the inhibitor arc (arc with a rounded head).

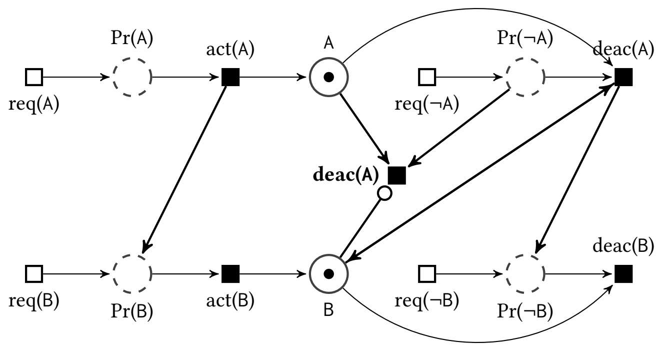

Figure 2: Context A is *causally* connected to context B

CoPNs are composed by means of two internal functions, cons and ext, and the specification of the context dependency relations between contexts. Effectively, these functions are used to extend the definition of the individual singleton CoPNs, by adding new transitions, inhibitors and arcs, specifying the behavior of the complete CoPN according the context dependency relation between contexts. In Figure 2, the new transitions and arcs defined by the causality dependency relation are shown between the singleton CoPNs in bold.

3 DISTRIBUTED CONTEXT PETRI NETS

Distributed Context Petri nets (DCoPNs) are defined as an extension of CoPNs to manage inconsistencies between contexts residing in different nodes, as they connect and disconnect.

Definition 3.1. A Distributed Context Petri net (DCoPN) is defined as a set of nodes PN, each containing a CoPN, together with a set of universal transitions UT, and the set of remote arcs and inhibitor arcs, F and $F_\circ$ respectively, connecting universal transitions to a CoPN, or connecting two CoPNs from different nodes. A DCoPN $\mathcal{D}$ is represented by the 4-tuple $\mathcal{D} = \langle PN, UT, F, F_\circ \rangle$.

The following describes the main building blocks and behavior of DCoPNs.

3.1 Composition and Interaction

To manage the dynamics of the system's context activation semantics, DCoPNs use a consensus algorithm based on the Raft consensus algorithm [9] (*cf.* Section 3.3). In this coordination model, there is a single *leader* node, responsible of managing the token game semantics across all nodes. All other nodes are *followers*, reporting their state and enabled transitions to the leader node. Note that to fire transitions with remote arcs, the transition must have access to the F and $F_\circ$ sets. The responsibility of firing transitions is delegated to the leader node, as this node is the only node with access to the universal transitions and remote transitions sets.

Interaction between DCoPNs happens as nodes connect and disconnect from the network. Upon connection, a new DCoPN is created from the composing nodes. Creation of DCoPNs follows directly from the composition of CoPNs and context dependency relations [4]. Whenever two CoPNs residing in different nodes PN_1 and PN_2 connect, the resulting DCoPN is the result of composing them according to the context dependency relations defined between their contained contexts, $\mathcal{D} = \langle PN_1 \cup PN_2, UT, F, F_\circ \rangle$. The resulting UT set is extended with the transitions managing the interaction between the CoPNs. Similarly, the F and $F_\circ$ sets are extended withe the remote arcs and inhibitor arcs added to connect the singleton CoPNs, and the universal transitions. Exactly which transitions and arcs extend the UT, F and $F_\circ$ sets, depends on the specific context dependency relations defined between the contexts in PN_1 and PN_2.

Figure 3 shows the example of two nodes, PN_1 containing the GPS context, and PN_2 containing the AgeGroup (AG) context. In this case, there is an implication dependency relation between the two contexts. The resulting DCoPN from the connection of the two nodes has two universal transitions, labeled deac(GPS) and deac(AG), composing the UT set (shown as the dotted oval in the figure). Correspondingly, six remote arcs are added to F (the regular arcs crossing the boundaries of a node, denoted as dashed lines in the figure), and four inhibitor arcs are added to $F_\circ$ (the inhibitor arcs with an end in the dotted oval in the figure).

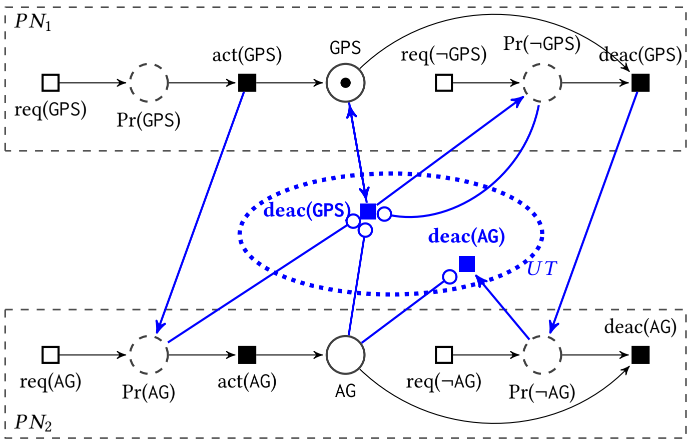

Figure 3: Two nodes interacting through an implication dependency relation

When a node disconnects from a DCoPN its contained CoPNs get removed from the PN set. That is, removing all singleton CoPNs in the node, the associated universal transitions in UT, and the arcs in the remote arc sets F and $F_\circ$ crossing the boundaries of a node. The removed CoPN can then connect with other DCoPN in the network, or become a DCoPN on its own.

As mentioned previously, the UT, F, and $F_\circ$ sets are accesible exclusively by the leader node. Any node in the DCoPN is elegible to be a leader, with the leader being dynamically updated as nodes connect and disconnect from the network. In Figure 3, for example, the PN_1 node could be selected as the leader; then, the node asks all other nodes for their transitions enabled to fire, starting from the transitions with highest priority. The leader then uses the information from the universal transitions UT and remote arcs F and $F_\circ$ to filter the set of enabled transitions, choosing one to fire at random. Transition firing is signaled from the leader to the follower nodes. The net marking in the DCoPN is updated based on the arcs in each CoPN in the follower node, and the remote arcs adjacent to the transition fired.

3.2 Consistency of Distributed Contexts

Inconsistencies arise due to the interaction between contexts. CoPNs propose a way to manage inconsistencies at run time through context dependency relations. In a distributed environment, inconsistencies may also arise from the connection and disconnection of nodes, as the set of context dependency relations between contexts changes dynamically. In particular, a DCoPN can reach an inconsistency if it presents an *unstable state* or a *conflict state.*

Definition 3.2. An unstable state in a DCoPN is a state in which there are temporary places marked, but no reactive transition can fire. This means that a context is preparing to activate or deactivate, but the associated action cannot take place.

Unstable states can be reached whenever, during the process of activating or deactivating a context, there is a node connection or disconnection. Given that the conditions in which transitions fire change upon the connection and disconnections of nodes (as the remote arc sets change over time), a transition enabled locally could become disabled globally, due to a remote arc. Later in Section 4 we show a concrete example in which a DCoPN reaches an unstable state in an exclusion dependency relation.

Definition 3.3. A conflict state in a DCoPN is a state in which two context places are marked, indicating the corresponding contexts as active, but the defined context dependency relations prohibit this state from ever being reachable.

Conflict states can take place upon nodes' connection or as the resolution of unstable states. For example, if two independent CoPNs are in a consistent state and their nodes connect, the consistency of the complete DCoPN could be compromised, as it is defined by the dependency relations between the two contexts. An example of such a case is described in Section 4 for the exclusion, implication, and requirement context dependency relations.

To manage the aforementioned inconsistencies, we add an additional verification step to the transition firing process as follows.

DCoPNs need to be able to revert to a consistent state from a detected inconsistency. To do this, we record the history of all consistent states in a node. After the firing process for all reactive transitions finishes (*i.e.,* no reactive transition is enabled), the leader queries the state of all follower nodes. In the case of unstable states, if a node responds with a marked temporary place, then the state of the node reverts back to its last consistent state —that is, before firing a transition modifying the node's marking.

Solving conflict state inconsistencies is a more involved problem, as we require to determine whether a state is reachable. The resolution process follows as shown in Figure 4. Whenever two nodes A and B connect (`conn(A,B)`), if there are active contexts (*i.e.,* one of the context places is marked), the leader queries all the nodes, related with the connecting node by means of a context dependency relation to build a conflict set. That is, two context places marked, where their marking violates the contract of the context dependency relation between them. For the contexts in the conflict set, their activation or deactivation is reverted by falling back to the last consisted state recorded. Exactly which contexts revert to the previous state, depends on the specific context dependency relation defined between the contexts.

Note that the inconsistencies resolution process is iterative, after an unstable or conflict state is resolved, the leader asks all nodes for their state to check for remaining inconsistencies. This processes finishes when there are no more conflict states identified.

3.3 Implementation

To implement DCoPNs we use the Go[1] programming language. The choice of Go is motivated by the concurrent and distributed computing capabilities of the language. Additionally, we use the noise[2] library for peer-to-peer and broadcast communication between

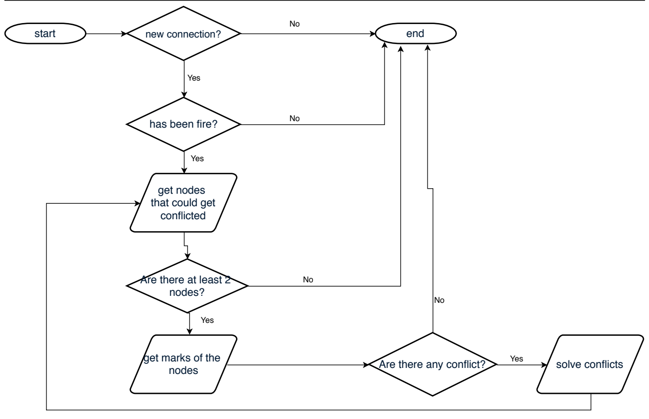

Figure 4: Conflict states resolution protocol

nodes. Finally, the implementation of the consensus algorithm follows a simplified version of the Raft algorithm.[3]

The implementation of DCoPNs is based on that of CoPNs, differentiated in four main characteristics. (1) Local CoPNs (*i.e.,* nodes) keep a complete history of their consistent states, (2) the set of universal transitions (UT), remote arcs (F), and remote inhibitor arcs ($F_\circ$) are global to the DCoPN and managed by the leader node, (3) there is a leader election algorithm to manage consensus, and (4) inconsistencies for the complete DCoPN takes into account unstable and conflict states.

3.3.1 Consistent States. CoPNs defined in a node are based on the `PetriNet` struct as shown in Snippet 1. Each Petri net is able to record the complete history of its consistent states by means of the `marksHistory` map, which is implemented as a last in-first out (LIFO) data structure. The state of a CoPN is recorded in the `marksHistory` map whenever the leader node sends a fire transition message to the node. While the CoPN is being modified, the state is kept as the current marking of each of its places (both temporary and context places). The CoPN's current state is precisely the state saved to `marksHistory` before the first request to modify its current marking, in the firing turn.

```
type PetriNet struct {
  ID int
  Context string
  transitions map[int]*Transition
  places map[int]*Place
  remoteTransitions map[int]*RemoteTransition
  maxPriority int
  marksHistory [] map[int]int
  universalPetriNet *petrinet.PetriNet
}
```

Snippet 1: Petri net structure for a node

3.3.2 Universal and Remote Sets. The UT, F and $F_\circ$ sets are added to the `PetriNode` structure for two main reasons. First, these sets allow to manage context dependency relations between CoPNs in different nodes. Second, unlike transitions and arcs that are

[1]golang.org
[2]https://github.com/perlin-network/noise
[3]The implementation of DCoPNs is available at: https://github.com/FLAGlab/DCoPN

part of a singleton CoPNs, the sets and their ownership change dynamically.

Upon creation, each context is defined with a `universalPetriNet` struct field. This field is composed of universal transitions, remote arcs and inhibitor arcs. Initially, this structure is undefined for all nodes. The structure is realized dynamically as the node is chosen as a leader. The universal transitions set UT is created based on the contexts and context dependency relations defined in the system. However, transitions in the UT set are not yet defined, as their endpoint may not be yet known —that is, they point to a remote context from which the address is unknown. A new leader adopts the UT set from the system's definition, and asks all other nodes which of their transitions are enabled to fired. The answer from each node, `[t, ctx, addr]`, contains the transition, the context, and the node address. Having the context and the address of the node, all the endpoints to universal transitions are resolved, creating the remote arcs F, and remote inhibitor arcs $F_\circ$ sets (according to the transition type). With the definition of these sets, transition enabling and firing can be verified.

3.3.3 Leader Selection. To manage the behavior of the DCoPN we use a consensus algorithm based on Raft. All behavior in the system is managed by a *leader* node. In our case, the leader is the node responsible of orchestrating transition firing, and detecting and solving inconsistencies. Throughout their life, nodes can be one of three types, *leader*, *follower*, or *candidate*.

Nodes in a distributed system can fail unannounced; it is possible that either a follower or the leader fail at run time. When a follower fails, if it reconnects, then it simply catches up with all requests sent by the leader. When the leader fails, the cluster of followers must realize it has no leader and select a new one by holding an election. To select a leader, some of the followers become candidates and assemble an election —that is, they ask all the other nodes to vote for a leader. Each node has exactly one vote per election term. In the end, the candidate with the majority of votes becomes the leader. The leader selection algorithm is described in Figure 5.

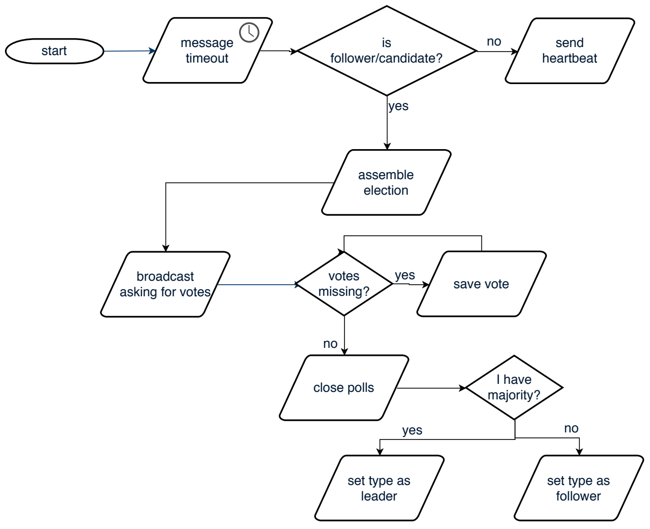

Figure 5: Leader election algorithm control flow

Leader communication is based on a Heartbeat (HB). When a follower receives a HB, it knows that the leader is still alive. Once a follower does not receive a HB in a timeout threshold, it begins an election.

Each node has a term —that is, a number establishing it as a viable candidate to become a leader. When an election is assembled, the node increments its current term and changes to a candidate state. It also broadcasts a message requesting votes to all other nodes. There are three possible outcomes of an election: the node wins the elections, another node establishes itself as leader, or a timeout occurs.

If the candidate wins the election it becomes the leader and broadcasts a HB to establish its leadership. If a candidate receives a message from another candidate claiming to be a leader, and the candidate's term is bigger than or equal to its current term, then it changes its state to a follower. Otherwise it ignores the message.

It is possible that several nodes become candidates at the same time making it that no candidate obtains the majority of the votes. In this case, after a given timeout every candidate becomes a follower and the process begins again. It is important to note that when a node changes its state it will always get a new random timeout. This behavior ensures that a leader will eventually get elected.

3.3.4 Inconsistencies Resolution. As mentioned previously, the leader node verifies the consistency of the DCoPN, whenever new nodes connect. The process to verify and solve the conflicts between two contexts from different nodes is shown in Snippet 2. For each of the contexts, based on their context dependency relations, a threshold values `ta` and `tb` are defined in Lines 10 and 11. Then, we go through all the nodes in which the context is defined (Lines 15-22). For each of the nodes, if the context markings are above the given threshold (Line 19), then the nodes are marked with the specific action required to solve the conflict, `ca` and `cb` respectively.

The resolution of the conflict (Line 30) takes the conflicts identified by the `getConflictAddrs` function, and applies the corrective action associated with the conflict and the node.

Note that the conflict resolution process is strongly dependent in the configuration of the context dependency relations. If new contexts or context dependency relations would appear at run time, these could modify the outcome of the algorithm.

4 VALIDATION

To validate the effectiveness of DCoPNs in managing the consistency of dynamically adaptive systems in distributed environments, we demonstrate their capacity to solve inconsistencies with respect to the interactions of remote contexts by means of the contexts dependency relations between them, in the context of the mobile city guide application.

Currently there are five main context dependency relations [2]. We now present each of the context dependency relations, together with the different situations (*i.e.,* connection and disconnection of nodes) in which inconsistencies (unstable or conflict states) can arise. We show the corresponding resolution strategy for each of such situations.

In the complete application we have three nodes with different contexts. The user node (PN_U), has the `LowBattery (LB)`, `GPS`, `GuidedTour (GT)`, a node for the museum (PN_M) containing the

```
func(cs *ConflictSolver) getConflictedAddrs(
marks map[string]map[int]*petrinet.RemoteArc,
ctx2address map[string][]string) map[string] bool{
   res := make(map[string]ConflictCommand)
   for _,value:=range cs.conflicts {
      ctxA := value.ctxA
      ctxB := value.ctxB
      pa := value.placeIdA
      pb := value.palceIdB
      ta := value.tokensA
      tb := value.tokensB
      ca := value.commandA
      cb := value.commandB
      if len(ctx2address[ctxA]) > 0 && len(
          ctx2address[ctxB]) > 0 {
         for _,addrA:=range ctx2address[ctxA] {
           marksA := marks[addressA][pa].Marks
           for _,addrB:=range ctx2address[ctxB] {
               marksB := marks[addrB][pb].Marks
               if marksA >= ta && marksB >= tb {
                  res[addrA] = ca
                  res[addrB] = cb
               }
            }
         }
      }
   }
   return res
}

func(pn *petriNode) solveConflicts() {
    conflictedAddrs := pn.conflictSolver.
        getConflictedAddrs()
    if len(conflictedAddrs) > 0 {
      for addrs, conflict:=range conflictedAddrs {
         pn.solve(addrs, conflict)
      }
       pn.step = CHECK_CONFLICTED_STEP
    } else {
       pn.resetStep() //state ok, go back to ask
       pn.didFire = false
       pn.needsToCheckForConflictedState = false
    }
}
```

Snippet 2: Conflict detection and resolution algorithm

`WifiConnectivity (WC)`, `AgeGroup (AG)`, and a bus node (PN_B) containing the `ItineraryPoiOrder (IPO)` context. As the nodes connect, any of them can become the leader in the DCoPN. Suppose the PN_U is the leader in the examples.

Exclusion. An exclusion dependency relation (A□−□B) involves two contexts, A and B, that are not allowed to be active at the same time; they could be inactive simultaneously. Two inconsistencies are identified from the interplay between contexts' activation and the connection of the nodes containing them.

Take the LB and WC contexts in their respective nodes. These contexts are in an exclusion dependency relation, LB□−□WC, as having a low battery prohibits the update of POIs and images download, while being in a wireless connection area promotes such behavior.

As the user has access to the museum, the nodes PN_U and PN_M connect. The leader node PN_U, verifies the consistency of its active contexts. In this case, if the user has a low battery upon entrance to the museum, both contexts LB and WC will be active. Furthermore, the dependency between the two contexts is created. In this situation the two contexts in the exclusion dependency relation are active, leading to a *conflict* state. As a consequence the two contexts' activations are *reverted* (`revert(WC)` and `revert(LB)`) to their previous consistent state.

Similarly, if the LB context is active, and the user's and museum's nodes connect as the WC context is preparing to activate (as shown in Figure 6), then the complete DCoPN is in an *unstable* state. To revert this situation, the activation of WC is reverted (`revert(WC)`).

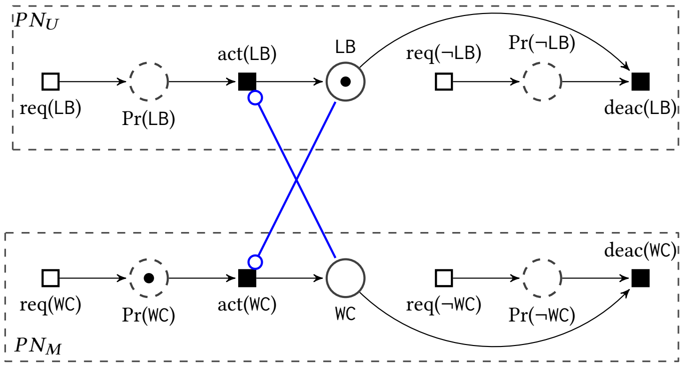

Figure 6: Unstable state when the `LowBattery` and `WifiConnectivity` contexts' nodes connect

Causality. A causality dependency relation (A−▷B) involves the activation and deactivation of context B whenever context A is activated and deactivated; while the activation and deactivation of context B does not have an effect on context A.

The causality dependency relation does not put a restriction in the state of the receiving context in the relation (*i.e.,* B). Therefore whenever two nodes interacting through a causality dependency relation connect, they will not reach any type of inconsistency.

Implication. An implication dependency relation (A−►B) involves the activation and deactivation of context B whenever A is activated and deactivated. In this case, the deactivation of B also triggers the deactivation of A.

As users in use of a GPS context walk into a museum, the AG context associated with the museum area is also activated to present the most relevant information about the area according to its target public. To represent this behavior, an implication dependency relation is used GPS−►AG. Consider the case in which a user has the GPS context active, but the AG context is not active in the museum (*e.g.,* due to an exclusive visit of a specific age group). As the nodes containing this contexts connect, the PN_U leader node will detect a *conflicting* state in the DCoPN, as shown in Figure 3. In such case, the activation of the GPS context is reverted (`revert(GPS)`).

Requirement. A requirement dependency relation (B−◄A) restricts the activation of the context B to context A being active. As a consequence, if context A becomes inactive, then B must be come inactive too.

To follow a guided tour visit of a city, users should organize their itineraries in a specific order. Therefore to activate the GT context, the IPO needs to be active first (GT−◀IPO). Consider the case in which a user starts a hop-on hop-off guided tour, with an active GT context in its node. However, as the users steps into the bus, his node containing the IPO context on his phone is still inactive. When the two nodes connect, a *conflict state* arises, as shown in Figure 7. To solve this inconsistency, the GT context reverts its activation (`revert(GT)`).

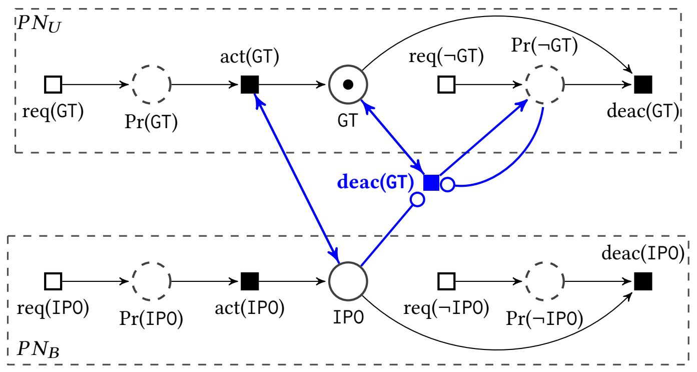

Figure 7: Conflict state when the `ItineraryPoiOrder` and `GuidedTour` contexts' nodes connect

Suggestion. A suggestion dependency relation (A - -▷B) involves a desire to see the behavior associated to context B when context A is activated. However, if context B cannot be activated, then the context A is activated anyways.

Similar to the case of the causality dependency relation, the suggestion dependency relation is a "best effort" dependency between the contexts. Therefore, there is no strong dependency between the active or inactive states of the contexts whenever their containing nodes connect. As a consequence no inconsistencies exist when using this context dependency relation.

Table 1 shows a summary of the inconsistency resolution strategies for context dependency relations. Based on the situations put forward for the interaction between nodes in a DCoPN we note that the disconnection of nodes does not lead to inconsistencies. This is because as nodes disconnect, the context depending relations between them are lost. Therefore, there is no binding interaction between the contexts restricting their state.

5 RELATED WORK

Distribution in adaptive software systems has been tackled from two distinct perspectives: programming languages definition and architectural approaches. Many of the existing approaches offer a verification mechanism to assure the consistency of the system in a given node. However, only one of the proposed approaches addresses the consistency of the systems across the boundaries of a computational node.

From the perspective of programming language, specifically COP languages, there are two approaches that tackle distribution, namely Lambic [14] and ContextErlang [11].

Lambic introduces predicate generic functions to manage context-base behavior in pervasive environments. The distribution model used in Lambic is based on the actor model, where use of remote functions takes place via message passing. Messages are applicable to specific implementations, so long they satisfy a particular predicate. Lambic, however, does not deal directly with the consistency between behavioral adaptations, therefore, if predicates on different nodes are applied to conflicting behavioral adaptations, the overall behavior of the system would be compromised.

ContextErlang is a context-oriented extension to Erlang's OTP distribution model. Behavioral adaptations are defined in ContextErlang as variations within an agent. ContextErlang manages inconsistencies between adaptations defined in an agent by means of the context Abstract Data Type (ADT). Context ADTs restrict the context activation to the allowed combinations declared in their specification. New contexts can be defined on an agent via remote variation transmission. However, to take these into account, the context ADT specification is required to change. Nonetheless, the context ADT model does not pose any verification of contexts' state across multiple agents; the assurance of coherence across agents is left to the developer in the specification of the context ADTs for each agent.

From the perspective of architectural approaches, three approaches targeting distributed systems that are suitable for the work presented in this paper: ACTRESS [8], FORMS [15], and FlashMob [13].

ACTRESS is a modeling environment for the development of self-adaptive systems using the Feedback Control Definition Language (FCDL). ACTRESS adaptations are realized by means of model transformations specified using a Domain-specific Language (DSL). ACTRESS offers a consistency verification process with respect to the FCDL specification, using an external analysis tool. This verification process is similar to that proposed with CoPNs. However, as CoPNs, it falls short in the verification of consistency for multiple nodes at once.

Similar to ACTRESS, FORMS presents a formal specification language for the definition of distributed self-adaptive systems. The reference model is used to define adaptations and relationships between them and their composition. The FORMS model is designed to help developers with the definition of self-adaptive distributed systems, together with a set of strategies to fix inconsistencies if they are ever present. The specification of such repair strategies is however restricted to interactions foreseen by developers before hand.

FlashMob uses a gossip protocol to derive consistent configuration of nodes, if one of them would ever fail. FlashMob is successful in adapting the system upon the failure or disconnection of a particular node. System recovery from node failure is done as part of the gossip protocol, in which a new (consistent) configuration of the system is found. Such behavior is the most closely related to the work proposed in this paper, as changes in the states of nodes, demand the evaluation of a reconfiguration for the complete system. The two solutions take advantage of distributed protocols to best

Context dependency relation	Situation	Problem	Solution
A□−□B	activate(A), activate(B), conn(A,B)	conflict state	revert(A), revert(B)
	activate(A), req(B), conn(A,B)	unstable state	revert(B)
A−▷B	-	None	-
A−▶B	activate(A), conn(A,B)	conflict state	revert(A)
B−◀A	activate(B), conn(A,B)	conflict state	revert(B)
A--▷B	-	None	-

Table 1: Causes of inconsistencies and their resolution in DCoPNs

find the next consistent state. The difference between the two approaches is the level of granularity in which the solution is defined. While FlashMob uses a configuration based on the available nodes, DCoPNs focus in the consistency of contexts within nodes.

6 CONCLUSION AND FUTURE WORK

This paper presents an extension to the CoPNs execution model to manage the consistency of distributed, dynamically adaptive software systems. To do this we extend the model with distributed systems' characteristics that enable inter-node communication. The proposed model is based on the Raft consensus algorithm to manage context activation form a global perspective. Additionally, the model is extended with an unstable and conflict resolution algorithms used to restore the system to a consistent state whenever inconsistencies yield form node connections. At the moment, the system implementation and conflict resolution algorithm is highly dependent of the defined context dependency relations. Our proposed model is validated relevant for the management of different remote nodes, using the conflict resolution algorithm to successfully identify and solve inconsistencies in the interaction defined by the different context dependency relations.

We identify three main directions in which our model could be extended or improved as part of the future work. To manage inconsistencies, the chosen solution was to revert to a consistent state form a particular CoPN. In the future, to make this process more robust, we would like to explore the effect of such rollbacks using different heuristics, as searching further in time, based in the point in which each of the contexts that generate the inconsistency were activated. Additionally, our solution depends on manual specification of conflicts, so a future improvement can be to develop ways to find such inconsistencies automatically and in a performant time.

The behavior to revert DCoPNs to their previous stable state can lead to situations in which the contexts is disregarded. For example in the case of the exclusion dependency relation inconsistency, the solution is to deactivate both contexts. Nonetheless, both situations are being sensed from the environment, which could lead to the reactivation of `WifiConnectivity` rather than `LowBattery`, leading to the depletion of the phone's battery. The decision of which context activation to revert could be based on contexts' priority.

Analysis algorithms to reason about system properties in a DCoPN should be explored as a part of a distributed extension of Petri nets. Based on these algorithms, it would be possible to further explore the effect of connection and disconnection of nodes, with respect to the previous results obtained about the DCoPN.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of the paper for their useful comments on earlier drafts of this document.

REFERENCES

[1] B. Bordel, R. Alcarri, T. Robles, and D. Martín. "Cyber–physical systems: Extending pervasive sensing from control theory to the Internet of Things". In: *Pervasive and Mobile Computing* 40 (2017), pp. 156–184.
[2] N. Cardozo. "Identification and Management of Inconsistencies in Dynamically Adaptive Software Systems". PhD thesis. Université catholique de Louvain - Vrije Universiteit Brussel, 2013.
[3] N. Cardozo, J. Vallejos, S. González, K. Mens, and T. D'Hondt. "Context Petri Nets: Enabling Consistent Composition of Context-Dependent Behavior". In: *Proceedings of the International Workshop on Petri Nets and Software Engineering.* CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 156–170.
[4] N. Cardozo et al. "Semantics for Consistent Activation in Context-Oriented Systems". In: *Journal of Information and Software Technology* 58.0 (2015), pp. 71–94.
[5] S. González et al. "Subjective-C: Bringing Context to Mobile Platform Programming". In: *Proceedings of the International Conference on Software Language Engineering.* Springer-Verlag, 2011, pp. 246–265.
[6] T. Kamina, T. Aotani, and H. Masuhara. "EventCJ: A Context-Oriented Programming Language with Declarative Event-based Context Transition". In: *Proceedings of International Conference on Aspect-Oriented Software Development.* AOSD'11. ACM Press, 2011, pp. 253–264.
[7] G. Kasin. "Engineering Context-Oriented Applications". MA thesis. Université Catholique de Louvain, 2012.
[8] F. Krikava, P. Collet, and R. France. "ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures". In: *Symposium On Applied Computing.* SAC'14. 2014.
[9] D. Ongaro and J. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the USENIX Annual Technical Conference.* USENIX'14. USENIX Association, 2014, pp. 305–320.
[10] M. Salehie and L. Tahvildari. "Self-Adaptive Software: Landscape and Research Challenges". In: *ACM Transactions on Autonomous and Adaptive Systems* 4.2 (2009), 14:1–14:42.
[11] G. Salvaneschi, C. Ghezzi, and M. Pradella. "ContextErlang: Introducing Context-Oriented Programming in the Actor Model". In: *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD'12).* ACM Press, 2012, pp. 191–202.
[12] G. Salvaneschi, C. Ghezzi, and M. Pradella. "Context-Oriented Programming: A Software Engineering Perspective". In: *Journal of Systems and Software* 85.8 (2012), pp. 1801–1817.
[13] D. Sykes, J. Magee, and J. Kramer. "FlashMob: distributed adaptive self-assembly". In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* SEAMS '11. ACM, 2011, pp. 100–109.
[14] J. Vallejos et al. "Predicated Generic Functions: Enabling Context-Dependent Method Dispatch". In: *Software Composition.* Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 66–81.
[15] D. Weyns, S. Malek, and J. Andersson. "FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems". In: *ACM Transactions on Autonomous and Adaptive Systems* 7.1 (2012).