Context Slices:

Lightweight discovery of behavioral adaptations

Nicolás Cardozo, Siobhán Clarke
Future Cities, DSG, Trinity College Dublin - College Green 2, Dublin 2, Ireland
{cardozon, siobhan.clarke}@scss.tcd.ie

ABSTRACT

Context-Oriented Programming languages enable the definition of systems that can adapt their behavior according to specific situations in their surrounding environment. Current approaches require developers to have prior knowledge about such situations and the adaptations applicable in each one. Such approach hinders the use of Context-Oriented Programming in modern open systems, in which the totality of the system may be unknown beforehand. We propose context slices which allow the autonomous discovery and composition of adaptations gathered from systems' surrounding environment. Context slices use zero configuration networking services to advertise and discover adaptations in the network, and an ontology structure to manage them. We show the applicability of context slices for cyber physical systems by means of an ambient assisted living scenario.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures, modules*

Keywords

Context-oriented programming, Context discovery, Internet of things

1. INTRODUCTION

The Context-Oriented Programming (COP) paradigm enables dynamic behavior adaptation of software systems with respect to their surrounding execution environment from a programming language perspective [7]. The vision of COP systems follows a reference architecture [4] consisting of four modules, *context discovery*, *context management*, *active context*, and *application behavior*. The main focus in the development of COP languages has been on the latter three to: (1) propose language abstractions allowing the definition of behavioral adaptations and their associated contexts, corresponding to situations in the system's surrounding execution environment, (2) study scoping mechanisms to dictate how and when behavioral adaptations are dynamically composed into, or withdrawn from the system. For example, proposing activation strategies to dynamically compose adaptations based on information gathered from the systems' surrounding execution environment, and (3) offer the most appropriate behavior of the system.

COP has proved relevant in the mobile computing domain by enabling dynamic behavior adaptations according to information gathered from the sensors available in mobile devices [4]. Using this model, developers are required to know beforehand all sensors from which the system can gather information. Contexts and behavioral adaptations are defined using such knowledge, making explicit which objects to adapt, when and where to enact adaptations, and the expected system behavior in all possible situations in the surrounding execution environment. This approach is unfeasible for the development of Cyber Physical Systems (CPSs) and the Internet of Things (IoT) as objects, sensors, or services are made available and unavailable unannounced in these environments. Therefore, it is not possible to define specialized behavior to adapt to such entities beforehand.

Furthermore, this approach requires the explicit definition of context objects for each new application. For example, in a domotics environment with a noise level sensor, a room environment application could define a `Meeting` context to detect noisy situations and increase the stereo's volume accordingly. This is shown in the following snippet using Context Traits [3], in which: contexts are defined as first-class objects using the `new` construct (Line 1), behavioral adaptations are defined as *traits* enclosing all behavior specializations (Lines 2 - 5), finally, behavioral adaptations and the objects they adapt are associated with a context using the `adapt` construct in Line 6.

```
1 Meeting = new cop.Context({name: "Meeting"});
2 NoisyMeeting = Trait({
3   adjustVolumeLevel: function() {
4     volume = 70%;
5   }  });
6 Meeting.adapt(Stereo, NoisyMeeting);
```

Similarly, a user personalization application, could define a `Meeting` context to set the advertising mode of a mobile phone to vibrate whenever the user is detected to be on a meeting, as shown in the code snippet below.

Context slices are put forward in COP languages as a means to enable the run time introduction of, and interac-

Preprint version.
COP Workshop co-located with ECOOP'15

```
1Meeting = new cop.Context({name: "Meeting"});
2SilentPhone = Trait({
3  advertise: function(num) {
4    console.log("Vibrating - call from" + num);
5  } });
6Meeting.adapt(Phone, SilentPhone);
```

tion with unknown adaptations as new sensing devices or services spontaneously appear on systems' surrounding execution environment. Doing so shifts adaptations' definition to sensors, thus avoiding the (re)definition of contexts for each new application.

Context slices consist of an ontology-based structure to manage contexts and their associated behavioral adaptations (Sections 2.1 and 2.3). Additionally, contexts' structured names are used to advertise/discover adaptations following a zero configuration networking protocol as the multicast Domain Name System (mDNS) (Section 2.2). In order to use context slices, we extended Context Traits to enable the definition, discovery, and communication of unknown contexts at the language level (Section 3). Context slices are shown applicable to CPS environments by means of an Ambient Assisted Living (AAL) in Section 4.

2. CONTEXT SLICES

Context slices are introduced as a lightweight discovery module for COP systems with the purpose of facilitating the integration of new services enriching software systems' behavior with respect to their environment. To meet this goal, the proposed discovery module builds on the ideas of zero configuration networking protocols for host name resolution, mDNS,[1] and ontology slices [5] as management structure.

The purpose of context slices is twofold. On the one hand, it provides a structure to advertise and discover contexts, avoiding their replication. On the other hand, it allows the system to restrict the discovery of contexts, such that only adaptations relevant to the system are discovered. The following presents how context slices enable autonomous discovery and composition of unknown adaptations.

2.1 Organizing Context Objects

Context slices consists of a lattice-like hierarchical structure for the categorization and management of contexts. In this structure contexts are grouped according to their common semantic purpose (*e.g.,* information source or type of gathered information), similar to contexts groupings in namespaces [3], or service grouping in semantic ontologies.

Context slices' lattices structure contexts similar to the way objects are hierarchically ordered through delegation relations ($\rightarrowtriangle$). Given the root c of a lattice L, L consists of all discovered contexts l, such that $l \rightarrowtriangle^* c$, where $\rightarrowtriangle^*$ denotes the transitive closure of the delegation relation. Using this construction of the lattice L, a *slice* S rooted at c is defined as $S := \langle c, L \rangle$. As an example suppose a COP system is defined to discover adaptations from five context domains, namely `resources`, `programming`, `environment`, `user`, and `services`. Figure 1 shows the five corresponding slices with all discovered contexts specializing each of them.

Contexts to be discovered by an application are determined by its developers according to the context domains that are semantically meaningful for systems' execution. The root of the five slices in Figure 1 provide examples of contexts domains that could be used in a COP system to organize adaptations. As part of the design of context slices, we propose a set of abstract context definitions capturing the *domains* and *sensors* to which a system could adapt taking into account the specific situations of its surrounding execution environment. Based on these, specific contexts can be defined further, providing behavioral adaptations with respect to *concrete*, *vendor*, or *specialization* contexts, as shown in the example of Figure 1. Note, however, that abstract context definitions are only suggested and are not readily provided as part of the context slices discovery module. Other domains and sensors could be defined by developers as new domains and sensing devices are introduced.

Context slices' lattices are generated upon context discovery (*cf.* Section 2.2). Whenever a context is discovered it is added to the lattice following its structured name from the root of the lattice. In the example of Figure 1 suppose there is no context discovered as part of the **user** domain and the `interview` context is discovered. This context is advertised with structured name `user.activity.meeting.interview`. The system will then create a slice $\langle$ `user`, `interview` $\rightarrowtriangle$ `meeting` $\rightarrowtriangle$ `activity` $\rightarrowtriangle$ `user`$\rangle$. If other context specializing the `user` domain is discovered, say `presentation`, all contexts part of its structured name not yet in the lattice are added to it. In this case, the `presentation` context is the only context added to the lattice, resulting in the slice $\langle$`user`, (`presentation` $\rightarrowtriangle$ `meeting`, `interview` $\rightarrowtriangle$ `meeting` $\rightarrowtriangle$ `activity` $\rightarrowtriangle$ `user`)$\rangle$. Note that context definitions are not replicated on the slice. Similar to uniqueness of objects in namespaces, contexts with the same structured name are assumed equal in context slices. That is, given three contexts c, c_1, c_2, if $c_1 \rightarrowtriangle c$ and $c_2 \rightarrowtriangle c$ then $c_1 \neq c_2$. As an example, take the two definitions of the `high` contexts in Figure 1. These definitions of the same context object are allowed as they do not specialize the same context. One context specializes context `memory` and the other specializes context `battery`. In contrast, all immediate specializations of either `memory` or `battery` must define different contexts.

2.2 Context Discovery and Advertising

CPSs consist of sensors, services, and physical entities that may be unknown to systems' developers beforehand. This is due to the scattered nature of the components in the system, or by the introduction of new sensors or services to the environment (*e.g.,* in IoT environments). Unknown entities may introduce new adaptations to the system. As a consequence, discovery of adaptations is vital to maintain the most appropriate behavior of a system with respect to its surrounding environment.

Context slices use the structured naming conventions defined by the mDNS protocol to autonomously advertise and discover contexts as part of the `cop` protocol (*cf.* Line 3 in Snippet 1 and Line 2 in Snippet 2). Structured names are paths defined by following the delegation links of the context until a root context definition is reached (*i.e.,* a context domain as in Figure 1). As contexts appear in a network their structured name is advertised. For example, the structured name to advertise the `meeting` context in Figure 1 is `user.activity.meeting`.

Contexts are advertised together with their associated behavioral adaptations. Whenever defined in external devices, adaptations are registered to the context manager by means

[1] `http://tools.ietf.org/html/rfc6762`.

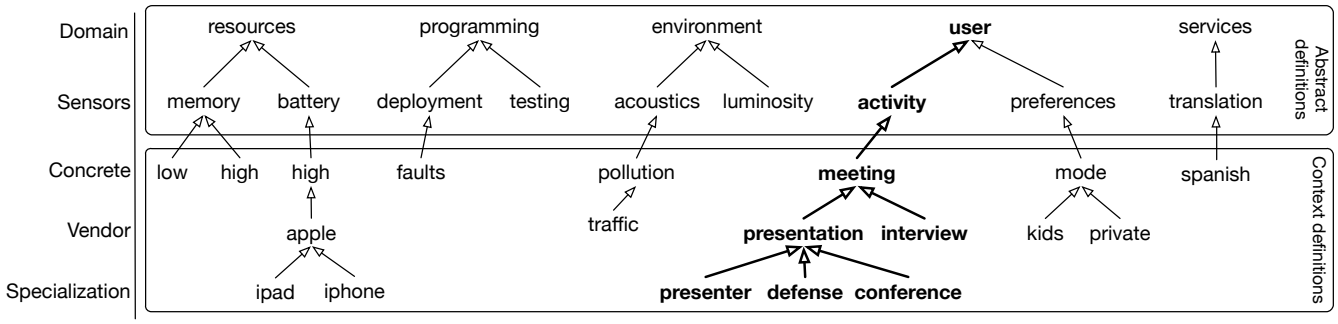

Figure 1: Lattice structure for five discovered slices in the system.

of the `registerAdaptation(ctx, adaptation)` construct. Registered adaptations are advertised upon network connection. For example, the `Meeting` adaptation is advertised with its context and associated behavioral adaptation as `registerAdaptation(Meeting, NoisyMeeting)`.

As adaptations dynamically appear in the surrounding environment of the system, their contexts are added to the context slices' lattice. New contexts are discovered if their structured name contains one of the slices browsed by the system. Contexts' browsing is defined in context manager using the `browse(slice)` construct. For example, browsing for the `user.activity` slice, results in the discovery of all (and only the) contexts in bold in Figure 1. If a context disappears from the surrounding environment (*e.g.,* due to a faulty sensor), it is removed from the lattice and its adaptations discarded. As with existing COP languages, removing behavioral adaptations while they are running leads to an error in the system, since the behavior is no longer defined. Contexts reappearing are treated as new contexts.

2.3 Scoping Adaptations

Systems do not have any prior knowledge about adaptations' implementation.Therefore, all system defined objects can be adapted by all discovered behavioral adaptations.

For every object o in the system, composition of all behavioral adaptations $T_{1_c}, \ldots, T_{n_c}$ associated to a discovered context c, results in the adapted object o' after the activation of context c —that is, $o' = o \cdot T_{1_c} \cdot \ldots \cdot T_{n_c}$. In the example of Section 1, the discovery of the `meeting` context as part of the `user.activity` slice introduces the `NoisyMeeting` and `SilentPhone` adaptations to all objects in the system (*i.e.,* `Stereo` and `Phone`). However, it may be desirable to restrict the contexts (and thus behavioral adaptations) that adapt an object in the system. To restrict adaptation composition to specific objects the `browse(slice, obj)` construct is used. The behavior provided by this construct is to compose the behavioral adaptations associated to the contexts specified by the `slice`, exclusively with the specified object, `obj`.

3. CONTEXT SLICES IMPLEMENTATION

Context slices focus on the autonomous discovery and composition of contexts in COP systems. Contexts slices' implementation consists of an ontology-based lattice structure and the network communication libraries to respectively manage, and advertise and discover contexts. However, developers do not directly interact with these functionalities. All interaction takes place through the context manager's API using the advertising and browsing constructs introduced in Section 2. Context Traits, a COP extension of JavaScript, is used as a concrete implementation artifact for context slices. Nonetheless, context slices could be introduced in a similar way as part of other COP languages.

3.1 Context-Oriented Programming

The core abstractions of COP languages are preserved with the introduction of context slices. Definition of behavioral adaptations, and context activation and deactivation remain as defined in Context Traits, respectively using traits, and the `activate` and `deactivate` constructs. Contexts' definition is extended to introduce the `slice` property used to advertise the context, as follows.

```
var Meeting = new cop.Context({
  name: "Meeting",
  slice: "user.activity"
});
```

3.2 Advertising, Discovery, and Communication

Context slices' implementation enables all COP systems to both advertise and discover contexts. For simplicity, hereinafter we refer to external devices to advertise contexts, and the system to discover them.

Advertising.

Adaptations are made available for discovery by registering their contexts and associated behavioral adaptations in the context manager. As shown in Snippet 1, adaptations advertisement consists of two steps. (1) Context's behavioral adaptations are stored in an `exportable` adaptations map (Line 2 in Snippet 1). (2) Adaptations in the `exportable` map are advertised autonomously upon connection of devices to a network. Adaptations are passed to the system via a fixed `socket.io` connection whenever they are discovered by the system. For each adaptation, all its partial methods are transmitted to the system as a *plain* JSON object (`json`), as shown in Lines 9 through 16 in Snippet 1.

Discovery.

The slices to discover contexts in the surrounding environment are defined as part of the system specification via the `browse` method. Contexts are discovered autonomously if an advertised context structured name contains the slices specified by the system. As adaptations are discovered, the context slices' lattice is updated (Line 10 in Snippet 2), keeping track of contexts and their associated behavioral adap-

```
registerAdaptation(ctx, adap) {
  Discovery.exportable[ctx.name].push(adap);
  var ad = mdns.createAdvertisement(mdns.tcp('
      cop'), port, {name: ctx.slice});
  ad.start();

  socket.on("conn", function(socket) {
    socket.emit("context", ctx);
    partialFunctions[ctx] = [];
    _.each(adap, function(method) {
      _.each(method, function(fun) {
        if(typeof(fun.value) === 'function')
          partialFuns[ctx].push(getNames(fun));
      });
    });
    json = {name: ctx.name, funs: partialFuns};
    socket.emit("adaptation", json);
  });
}
```

Snippet 1: Adaptations' advertising.

tations. Whenever a device defining adaptations disappears from the network, the context objects and their associated behavioral adaptations are removed from the lattice (Line 23 of Snippet 2).

```
browse(slice) {
  browser=mdns.createBrowser(mdns.tcp('cop'));
  browser.on('serviceUp', function(service) {
    socket.emit("conn", "connected");
    this.connect(slice);
  });
}
function connect(slice) {
   socket.on("context", function(data) {
      lattice.append(new cop.Context({name:
          data.name}));
   });
   socket.on("adaptation", function(json) {
      var obj = {};
      _.each(json, function(pb) {
         obj[pb] = function() {
            socket.emit("remote", [pb,
                arguments], callback);
         };
      });
      var proxy = new Trait(obj);
      lattice.addAdaptation(name, proxy);
   });
   socket.on("disconnect", function(name) {
      lattice.remove(name);
   });
}
```

Snippet 2: Adaptations' discovery.

Note that behavioral adaptations are not passed to the system *as is*. Given that behavioral adaptations are defined by third parties, their behavior is unknown to the system, presenting a potential threat to the system's integrity. To avoid harmful behavior to be deployed in the system, incoming behavioral adaptations are enacted by system created proxy objects, `proxy` in Line 19 of Snippet 2. Proxies provide the functional API of discovered behavioral adaptations, without incurring in the risks of copying their functionality into the system. Proxies creation is shown in Lines 12 through 20 of Snippet 2 where a new trait object (`proxy`) is created from the `json` object passed to the system. The functional API of the trait is given by the behavioral adaptations `pb` passed in the `json` object.

Communication.

Message exchanges between the system and external devices take place via fixed socket connections, established by the system upon discovery of contexts (Lines 3 through 5 in Snippet 2). In particular, method calls to behavioral adaptations are forwarded to external devices as specified by the proxy object, shown in Line 16 of Snippet 2. Figure 2 shows the interaction between the system ($SysEnv_1$) and two external devices ($CtxEnv_1$, $CtxEnv_2$). Whenever an adaptation is active (*i.e.,* its behavioral adaptations are composed in the system), all calls to the partial methods defined in the proxy object are forwarded to the external device using the *object reference* socket connection. Results of method calls are retrieved to the system via callbacks (Line 16 in Snippet 2). Furthermore, Figure 2 shows that multiple defined contexts in different external devices are not replicated in the system; a fixed connection is created for only one of the discovered context definitions.

The current communication model of context slices does not allow the communication of functions from external devices to the system. This is due to the limited support for communicating functions using sockets. Partial behavior in the external devices is not passed to the proxy object. These functions are called from proxies without knowing the internals of their specification. Therefore, `proceed` calls in the external device would be lost, as external devices do not have information about other available contexts. Generation of a smarter proxy API harnessing the power of callbacks to deal with proceed calls is part of our future work.

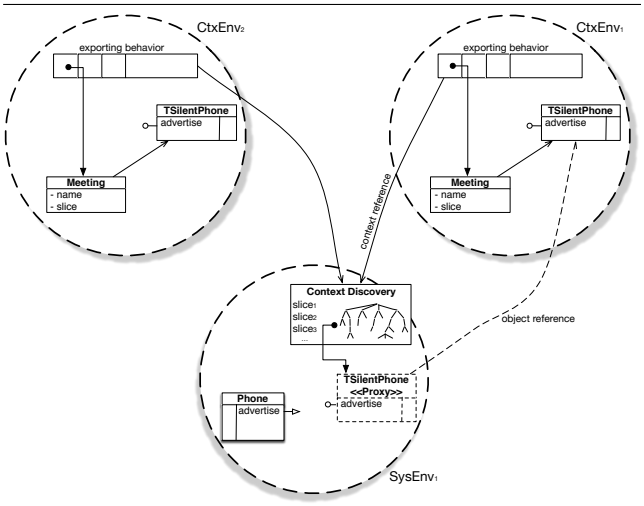

Figure 2: Communication scheme for the discovery module in context traits.

3.3 Behavior Composition

Composition of behavioral adaptations in the system is driven by the (de)activation of contexts with respect to the information gathered about the system's surrounding execution environment from external devices. Along side adaptation's definition, external devices also define the logic to activate and deactivate their respective contexts based on sensor information. Context activation and deactivation are implemented, for example, using expert systems which send a remote message to (de)activate a context in the system. These messages are then received by the system, and the associated

context is (de)activated as shown in Snippet 3. Once contexts are activated/deactivated, the proxy representations of their associated behavioral adaptations are composed into/withdrawn from the system as specified in COP systems.

```
//(de)activation messages sent from sensors
socket.emit("activate", ctx.name);
socket.emit("deactivate", ctx.name);
//System context activation and deactivation
socket.on("activate", function(ctx) {
   lattice.activate(ctx);
});
socket.on("deactivate", function(ctx) {
   lattice.deactivate(ctx);
});
```

Snippet 3: Context (de)activation interaction.

4. CONTEXT SLICES IN ACTION

Interaction between a system and its possible adaptations as proposed in the context slices discovery module matches the vision of CPSs and the IoT. These systems are composed of multiple distributed entities (*e.g.,* sensors, robots, agents) interacting to provide a specific service. A key characteristic of such systems is that they are plug & play —that is, entities may appear or disappear unannounced, seamlessly integrating new behavior, or accommodating to disappearing behavior. We developed a simple prototype AAL application as representative of such systems.

AAL systems consist of people interacting with different monitoring sensors (*e.g.,* cameras, motion sensors), and body area networks to monitor their activities and health. Gathered user information can be linked to a mobile device and displayed to users; alternatively measurements can be sent directly to caregivers or attending doctors to report users' status. As a particular example, consider a wellness program composed of sensors for measuring pulse rates and motion activity of a user. Such information is used by the system to propose a series of activities the user should follow as part of the program. Suppose the initial behavior of the application is to suggest the user to run for 20 minutes every day, as shown below.

```
activity = function() {
  alert('Use treadmill for 20 mins');
}
```

As part of the wellness program the system is open to all services declared within the program by allowing the discovery of adaptations in the `services.wellness` program, browsing for such slice. Additionally, the system enables the discovery of user defined adaptations by browsing for a `user` slice. These slices are declared on the system by calling the context manager's browse method are as shown below. Using `browse` the system actively listens for new advertised entities corresponding the specified slices.

```
cop.manager.browse("services.wellness");
cop.manager.browse("user");
```

We test two behavioral adaptation scenarios appropriate for the discovery of adaptations using context slices: discovery of previously advertised contexts, and introduction of new contexts to running systems.

Suppose that in addition to treadmills, the AAL environment is equipped with a static bike offering alternative training for users who suffer from jumpers knee. Such condition can be detected via a body sensor gathering knees' strain information. Whenever strained knee conditions are detected, the application should propose warm-up and cooldown activities in addition to alternative exercise activities (*e.g.,* using the static bike). The `KneeStrain` context is readily available and advertised as part of the wellness program.

```
KneeStrain = new cop.Context({
 name = "KneeStrain",
 slice = "services.wellness"
 });
StrainedKnee = Trait({
  warmup: function() { ... },
  activity: function() {
    warmup();
    alert('Use static bike for 20 mins');
    cooldown();
  },
  cooldown: function() { ... }
});
registerAdaptation(KneeStrain, StrainedKnee);
```

The context and its associated behavioral adaptations are defined by the developers of the body area network devices. The context is advertised as specified by its `slice` property —that is with slice `services.wellness.KneeStrain`. The context will be discovered by the system as it is browsing for all contexts containing the slice `services.wellness`.

Suppose now that during the training period the user decides to run a marathon. An external personal trainer application can interact seamlessly with the wellness program if this advertises its services, for example, using the `user` slice. The marathon training intensifies the workout of users as time passes by, according to their preferences, as shown in the code bellow.

```
Marathon = new cop.Context({
  name: "Marathon",
  slice: "user.activity"
});
Training = trait({
  activity: function() {
     alert('Use treadmill for 10km');
  }
});
registerAdaptation(Marathon, Training);
```

Whenever the `Marathon` context is discovered, the slice `user.activity.Marathon` is added to the system's lattice. The `Training` behavioral adaptation associated with the context is composed into the system, modifying the activity in the wellness program. When the marathon passes the default preferences are reinstated, removing the slice from the system and reverting to the original application behavior.

In both scenarios we observed adaptations are autonomously discovered using context slices, effectively modifying the lattice structure. Moreover, behavioral adaptations are successfully composed in the system whenever contexts are signaled for activation, without the system or external devices requiring previous knowledge about each other.

5. RELATED WORK

This section presents approaches related to the context slices proposal classified in two categories according to the features they provide, that is, context clustering, and ad hoc discovery and service composition.

Context clustering in COP languages.

As previously mentioned, COP languages do not currently support autonomous discovery of unknown contexts. How-

ever, existing languages provide constructs to scope behavioral adaptations based on defined groupings of contexts.

Context slices' definition is inspired by context clustering using namespaces, already provided in Context Traits [3]. Namespaces and contexts belonging to them need to be explicitly specified beforehand by Context Traits developers.

In Flute [1] behavioral adaptations (*modes*) are grouped into *modals*. Modals are used by the run-time environment to ensure that the execution of modes entirely takes place under the correct context conditions. Similar to context slices, developers are not required to know all modes belonging to a modal beforehand, but modes can be added dynamically to modals. Nonetheless, Flute does not provide the discovery of modes or modals, both need to be defined by developers in advance.

Lambic [8] introduces *futurized generic functions* to adhere to the ambient-oriented paradigm. Via futurized generic functions, objects in the system are advertised and discovered using *tags*. In Lambic contexts are defined as predicates associated with the functions they intend to adapt, therefore, advertisement and discovery of objects and their methods implies context advertisement and discovery. Lambic additionally defines *group generic functions*. These functions are used to orchestrate the distributed and adaptive behavior of objects sharing similar functionality. The discovery model in Lambic is closely related to that of context slices, providing discovery of unknown adaptations via tags, and grouping related objects by means of group behavior.

Service discovery and composition.

In mobile ad hoc environments, network connectivity is volatile and objects may appear and disappear from the network unannounced. *typetags* and *remote references* are proposed to enable discovery and interaction between distributed event-loops [9]. Typetags are used to discover specific objects or services broadcasted to the network. Far references keep a casual connection between system entities to enable their communication. This communication model is similar to the advertise/discover one used in context slices.

Service composition mechanisms [2, 6] use ontologies to generate links between services. Such links are characterized by the semantic similarity between services. If the functionality provided by two services is semantically related, then a link between the two is generated. Semantic links are maintained on a (centralized) graph structure where services are represented as graph's nodes. Links are generated between nodes as new services appear in the network. Context slices are inspired on the semantic ontology graphs of service composition. Context slices take the ontology graph idea from service composition to manage discovered services, coupling it with ad hoc network discovery. We extend the ontology graph model with scoped discovery of available services, this enables us to focus on those services relevant for the system rather than on whole services graphs.

6. CONCLUSION

This paper introduces lightweight discovery of contexts in COP by means of context slices. Context slices use the mDNS protocol to enable the discovery of ad hoc adaptations advertised by external devices and sensors. Such devices gather information from the system's execution environment and define specific situations in which the behavior of the system can be adapted, according to defined adaptations (*i.e.,* context objects and their associated behavioral adaptations). As adaptations appear and disappear in the network, a lattice is used in context slices to manage the system's adaptations coming form external devices. The work on context slices is of significant value for CPSs and IoT environments, where new devices and services may appear after the system has been deployed. Context slices are the first step to fully realize the vision of such systems. In this paper we showed the applicability of context slices to CPSs by means of an AAL application prototype.

Contexts slices currently enable the discovery of adaptations corresponding to the slices specified by developers. An extension of the model would enable the discovery of all possible adaptations in the network without incorporating them in the system. Users then will choose those adaptations that are relevant to their surrounding environment. As mentioned previously, another avenue of future work for contexts slices is to manage proceed calls from the proxy objects created upon discovery of adaptations.

Acknowledgements

This work was supported by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Center (`www.lero.ie`). We thank the reviewers for their comments on earlier versions of this paper.

References

[1] E. Bainomugisha, J. Vallejos, C. De Roover, A. Lombide Carreton, and W. De Meuter. *Interruptible Context-dependent Executions: A Fresh Look at Programming Context-aware Applications*. Symp. on New Ideas and Reflections on Software. ACM, 2012.

[2] D. Bianchini, V. D. Antonellis, and M. Melchiori. *P2P-SDSD: on-the-Fly Service-Based Collaboration in Distributed Systems*. Journal Metadata Semantic Ontologies 5.3 (2010).

[3] S. González, K. Mens, M. Colacioiu, and W. Cazzola. *Context Traits: Dynamic Behaviour Adaptation Through Run-Time Trait Recomposition*. Intl. Conf. on Aspect-oriented software development. 12. ACM, 2013.

[4] S. González et al. *Subjective-C: Bringing Context to Mobile Platform Programming*. Intl. Conf. on Software Language Engineering. Springer-Verlag, 2011.

[5] L. Jin and L. Liu. *An Ontology Slicing Method Based on Ontology Definition Metamodel**. Business Information Systems. Springer-Verlag, 2007.

[6] S. Montanelli et al. *The ESTEEM platform: enabling P2P semantic collaboration through emerging collective knowledge*. Journal of Intelligent Information Systems 36.2 (2011).

[7] G. Salvaneschi, C. Ghezzi, and M. Pradella. *Context-oriented Programming: A Software Engineering Perspective*. Journal of Systems and Software 85.8 (2012).

[8] J. Vallejos. *Modularising Context Dependency and Group Behaviour in Ambient-oriented Programming*. PhD thesis. Vrije Universiteit Brussel, 2011.

[9] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedeker, and W. De Meuter. *AmbientTalk: Object-oriented, Event-driven Programming In Mobile Ad-hoc Networks*. European Conf. in Object-Oriented Programming. 2007.