ComPOS: Composing Oblivious Services

Alfred Åkesson, Görel Hedin, Mattias Nordahl, Boris Magnusson
Lund University, Sweden
{alfred.akesson,gorel.hedin,mattias.nordahl,boris.magnusson}@cs.lth.se

***Abstract*—Future internet-of-things systems need to be able to combine heterogeneous services and support weak connectivity. In this paper, we introduce ComPOS, a new domain-specific language for composing services in IoT systems. We show how Maria, a bird watcher, can use ComPOS to build a system that allows her to spy on birds in the garden while she is not at home. We demonstrate how ComPOS handles the unpredictable nature of IoT system by analysing in what cases Maria's system is still useful when some devices are unavailable.**

***Index Terms*—DSL, IoT, service composition, end-user programming**

I. Introduction

Current IoT applications typically have a cloud-centric architecture, where sensor devices stream data to cloud servers, computation and storage takes place in the cloud, and user applications interact with the data in the cloud. This architecture leads to IoT platform silos, that work in isolation. However, this makes it difficult to compose existing data, services, and devices from different silos into new applications [1], [2], [3]. Also, future IoT applications are expected to contain more powerful devices, with more computation taking place at the edge of the network, and need to handle unreliable (weak) connectivity of heterogeneous networks in a robust way [4].

We are exploring how to program such new kind of systems. Our goals are to support flexible integration of heterogeneous services to avoid the current silos, and to support the programming of robust applications that continue to work partially even if connectivity is temporarily lost.

Our approach is based on the Palcom IoT architecture [5], [6], [7] which uses asynchronous message passing between services hosted on devices. There are two main kinds of services: *native* services that contain computations and interaction with the physical world, and *composition* services that compose native services into applications, mediating and adapting messages between them. Native services are *oblivious*, meaning that they don't set up any connections to other services, and they don't necessarily know the identity of the service and device at the other end of a connection. This makes them reusable in different applications. Compositions, on the other hand, define which oblivious services on which devices that should be composed, and how messages are mediated and adapted.

Metaphorically, we can think of native services as ports on physical devices, and compositions as multiway adaptor cables that connect these ports. Additionally, a composition may itself have oblivious services, so called *synthesized* services, that other compositions can connect to. This would correspond to there being a port on the adaptor cable that another cable can plug into.

Metaphors from the physical world often need to be enhanced with some degree of "magic" to better fit a computational system [8]. In our case, the multiway cable (composition) has the ability to automatically connect itself to devices, as soon as they are within reach and turned on. To have this capability, the composition is itself hosted on a device, and "within reach" means that the two devices can reach each other via some network. Furthermore, each "cable end" of a composition can adapt to fit in the "port" of the oblivious service, so they do not depend on specific standardized service interfaces. Additionally, it is possible for several different "cables" to connect to the same "port" at the same time.

Figure 1 shows a conceptual model for devices and services: *services* are hosted on *devices*. Services can be either *oblivious* or *compositions*, where a composition connects to zero or more oblivious services. An oblivious service can be either *native* or *synthesized*, the latter being part of a composition. Each oblivious service has an *interface* of incoming and outgoing messages.

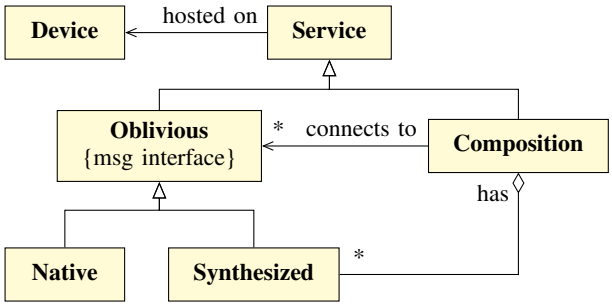

Fig. 1. Conceptual model for Palcom devices and services.

This paper presents a domain-specific language (DSL) for programming compositions. The language, ComPOS (Composition language for Palcom Oblivious Services), generalizes the currently used Palcom composition language [5], that was too simple for many interesting applications. In particular, ComPOS supports nested and parallel message sequences, and request messages that may have alternative replies. To support these constructs, ComPOS introduces *activations* that add state to the compositions. A main design goal is to make the language very easy to use, as well as to allow analysis with respect to composition. For this reason, ComPOS includes only constructs related to messages and message sequencing, and all computations on data are delegated to native services.

978-1-5386-9151-9/19/$31.00 ©2019 IEEE

In the following sections, we first present a motivating example for composing services (Section II). We then introduce our DSL, explaining the features and the interpreter (Section III). To exemplify more features of the DSL we do a couple of extensions to the example (Section IV). We then introduce a *utility analysis* in order to analyze what happens in the system when devices come and go. We use this analysis to analyze the extended example (Section V). Finally, we discuss related work (Section VI), and end with conclusions and prospectives for future research (Section VII).

II. Motivating example

As a motivating example, we will use a variant of a bird watching scenario [9], and discuss how to construct a supporting application using ComPOS.

A. Bird watching scenario

Maria is interested in birds and likes to keep track of what birds visit her garden. However, she cannot constantly be on the watch, so she would like to have an automatic system that does it for her. She has an idea of building a system that will automatically take pictures of the birds during the day, which she can check when she gets home. She has hardware and software that she wants to use to build the system: a motion sensor, a camera, and some artificial intelligence software that can recognize if a bird is present in an image or not. She would like to design the system such that it takes a photo every time the motion sensor detects that something is moving in the garden. If the bird recognition software detects a bird in the photo, it should get saved for later inspection.

B. Devices and services used

Figure 2 shows the hardware and software that Maria uses to implement the automatic bird watcher application. The camera, the motion sensor, and the laptop are devices connected to the local wifi network and they can discover each other using the Palcom middleware.[1]

The functionality is packaged as Palcom native services, each exposing an interface, specifying the messages that the service can send and receive. A message can be a *command* (not expecting a response), a *request*, or a *response* to a previous request. For a given request, like *has_bird*, there can be several alternative responses, like *bird* and *not_bird*. Messages can have parameters to transfer data between services. For example, the *has_bird* request has a parameter *img* for the image to be analyzed.

The services in Maria's system are:

- A storage service, to which images can be sent.
- A bird service that can classify an image as containing a bird or not.
- A motion service that sends a move command each time a movement is detected

[1]The Palcom middleware allows automatic discovery of devices and services on application-defined networks consisting of local UDP networks, connected using UDP or TCP/IP.

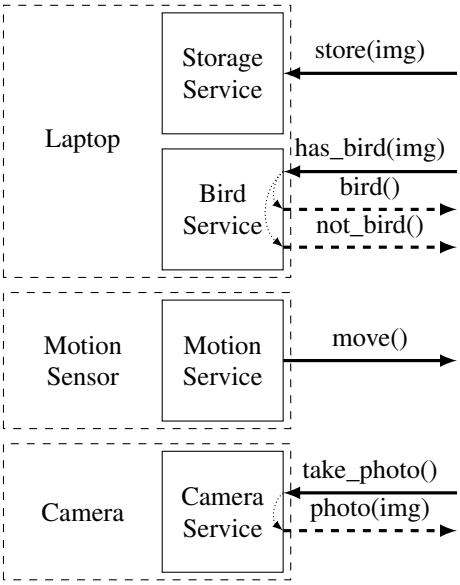

Fig. 2. Services (solid boxes) running on devices (dashed boxes). Commands and requests (solid arrows), responses (dashed arrows).

- A camera service that can take a photo on request and return the image.

Commands and requests are said to be *spontaneous* messages. When a spontaneous message is received, it starts a new independent *reaction* in the receiving service. I.e., spontaneous messages are not considered to have any causal relationship to previously received messages. Responses, on the other hand, are *expected*, and will continue a reaction that was initially started by a spontaneous message.

C. Composing the application

To construct the bird watcher application from the above services, Maria creates a *composition* (a ComPOS program), that connects to the relevant services, and that includes a script for how messages should be mediated. Figure 3 shows the composition script and a corresponding sequence diagram: When a *move* message arrives from the motion sensor, a *take_photo* request is sent to the camera, which responds with a *photo* message. A request *has_bird* is then sent to the bird recognition service which responds with either a *bird* or a *not_bird* reply. In the case of a *bird* reply, a command *store* is sent to the storage service. In addition to the script, the composition contains a configuration part that lists what services are used, what devices they run on, and what local names are used for these services (not included in figure 3). Both the configuration part and the script can be created in an easy way using structure editing and drag-and-drop from a service discovery browser [7], [10].

Maria deploys the composition service to the Laptop device and starts it. The system now store images of the birds during the day and she can come home after work and enjoy a new set of bird photos.

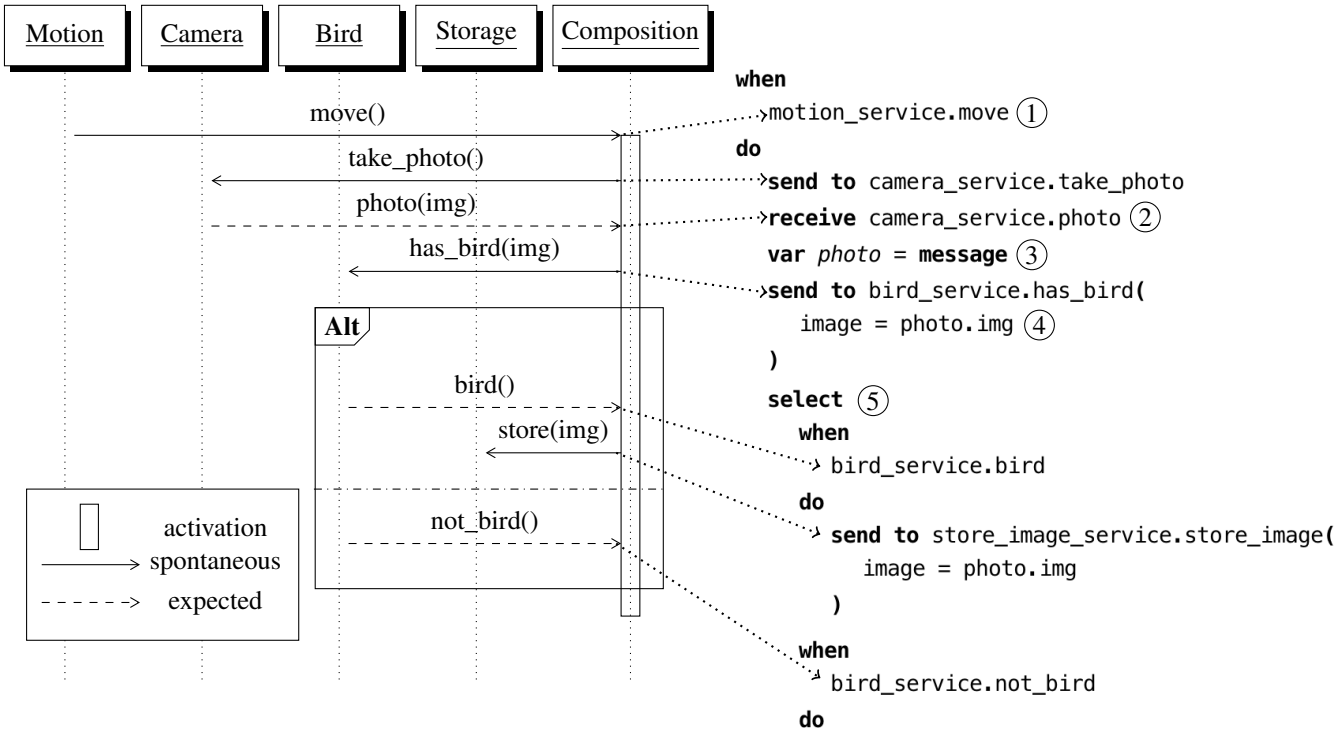

Fig. 3. Bird watcher composition script (right) with corresponding sequence diagram (left). Dotted arrows indicate what part of the sequence diagram corresponds to what part of the code.

III. The ComPOS language

A. Coordination constructs

A ComPOS coordination script consists of a set of guarded actions, so called *when-do*s. The *when* part contains an input action with a sender id and a message name, see (1) in Figure 3 for an example. The input actions must be mutually exclusive, so when a message arrives, there is only one guarded action that can match. The *do* part, also called a *reaction*, contains a sequence of actions to be executed when the input message is received.

Actions can be *blocking* or *non-blocking*. *Assignment* is a non-blocking action that assigns a value to a local variable. The value can be a literal, a reference to another variable, the latest received message (using the *message* keyword (3)), or a dereference for accessing a part of a structured value, for example, a parameter of a message (4).

The *send* action is non-blocking, and sends a command or request message to a receiver. Arguments to the message are assigned in a similar way as variables.

The *receive* action is blocking, and waits for a response from a previous request (2). The *select* action is also blocking, and contains a set of mutually exclusive guarded actions (5), like at the top level of the script. However, in a *select*, the input actions are responses, whereas at the top level, they are commands or requests.

There are also actions *parallel* and *finish first* that both run a set of action sequences in parallel. The difference is that *parallel* will block until all action sequences are finished, whereas *finish first* will block until one of the sequences has finished. These actions will be exemplified in Section IV.

B. The ComPOS interpreter

To execute ComPOS scripts, we implemented an interpreter. When the interpreter starts running a composition, it sets up connections to all currently discoverable services that are specified in the configuration part of the composition. During interpretation, connections are automatically set up or taken down, as the corresponding remote services are discovered or undiscovered, e.g., due to network errors. The interpreter has an event queue for incoming messages, and handles events in order of arrival. When receiving a message that matches a when-do clause at the outermost level, the interpreter creates a new *activation* for the corresponding reaction. The activation is associated with the connection for the incoming message starting the reaction. It contains values for local variables and keeps track of the currently executing action. The interpreter continues to execute the reaction of the current activation until it blocks or finishes. If the reaction blocks, the activation is suspended, and the interpreter continues by processing the next message in the event queue. If the next message is a response expected by a suspended activation, the interpreter continues to execute that activation. A received message can match at most one outer when-do, or blocked activation, and gets ignored if it has no match. Messages sent to connections that are currently down, are by default lost.[2]

[2]There is also functionality for declaring connections as *reliable*, in which case the messages are buffered until the connection is up again.

C. Semantics of incoming messages

A blocked activation will continue to run when the message it is waiting for arrives. However, if a connection is temporarily down, or if the remote service is not working as expected, this might take a very long time, or might never happen at all. It might also be the case that new spontaneous messages arrive on the same connection, in which case it is unclear if this should start a new reaction or not. For example, what should happen if a new *move* message arrives, while there is already an ongoing reaction for an earlier *move*?

Some possible ways of dealing with this situation are

1) start a parallel reaction for the new message
2) queue up the message and start its reaction when the ongoing reaction has completed
3) ignore the new message
4) abort the current reaction and start a new one

In our interpreter, we have chosen solution 4: abort the current reaction, and start a new reaction for the new message. This way we avoid old activations that remain indefinitely, which would happen in solutions 1, 2, and 3, and we avoid having an unbounded number of simultaneous activations that might arise from solution 1. Furthermore, solution 4 will prioritize the latest information, in contrast to solutions 2 and 3. [3]

D. Separating activations related to remote reactions

Sometimes, incoming spontaneous messages are independent, and should not abort each other's activations. For example, if two messages sent from different parallel branches trigger the same reaction in a remote service, those remote reactions are independent. Therefore, an activation is associated not only with the connection of its initial message, but also with a *reaction id* of the remote service that sent the message. Messages on the same connection but with different reaction ids result in independent activations that do not abort each other. Different branches in a parallel action are viewed as sub-reactions, and the interpreter automatically assigns different reaction ids to them. This allows for having multiple ongoing activations associated with the same connection. An example of this is given in Section IV-B.

IV. Extending the bird watcher scenario

We will now extend the bird watcher scenario to illustrate the use of synthesized services and multiple remote reactions.

A. Composing compositions using synthesized services

A synthesized service is an abstraction mechanism that allows a composition to provide functionality in the form of oblivious services [6]. This means that a composition can coordinate multiple services and provide their combined functionality as an oblivious service for the rest of the system. For example, if Maria finds out that the bird service gives too many false negatives, she may want to combine her local bird service with one she finds online. She decides to create a composition, *SynthBird*, with a synthesized service that has the same interface as the bird service but is a combination of a local bird service and an online bird service. It replies *bird* if either of the local or the remote bird services recognizes a bird in the picture, and *not_bird* if both the local and the remote bird services reply *not_bird*, see Figure 4 (right). Messages received by the synthesized service of a composition can be used as guarded actions in the when-dos, and reactions can send and reply messages to other compositions connected to its synthesized service.

B. Reactions triggered from multiple remote reactions

Suppose now that Maria wants to add one more camera to her system, to see the birds from more angles. She modifies her composition and uses the parallel action to allow both cameras to take and process pictures in parallel. We call this modified version of the composition *TwoCams*, see Figure 4 (left). This composition uses the synthesized service SynthBird to use the combined local and remote bird-recognizing services.

The reaction ids are embedded in the request messages sent to the SynthBird composition, allowing it to differentiate between the different requests. This way, the SynthBird composition will create one reaction for each request, and send responses back to the appropriate branch.

V. Utility Analysis

One requirement of the composition language was to support mobile devices and weak connectivity. In this section, we introduce a *utility analysis* to analyze what happens to the utility of the system when devices come and go. We will apply this analysis to the system shown in figure 4. The utility analysis looks at what happens if one or several of the devices disconnect from the others for some reason. This is to emulate what happens if a device is, for example, out of reach, out of battery, or has connection problems[4].

The purpose of Maria's system is to store images of birds for later inspection. For the purpose of utility analysis, we say that the system is *useless* if no image can get stored, due to some devices being disconnected, and that the system is *useful* if images can in some way get stored. The utility analysis explores whether the system is useful or useless when different sets of devices are disconnected. Table I shows the utility analysis of the system in 4.

From Table I we see that having a reaction being aborted when a new *move* command arrives, allows the system to still be useful when devices come and go.

VI. Related work

A. Previous composition languages for Palcom

Svensson Fors [5] introduced compositions (called *assemblies*) and synthesized services in Palcom. These compositions are stateless, limiting reactions to only contain actions for sending messages and setting global variables. Svensson Fors' implementation is included in the current release of Palcom

[3] There might be situations where options 1, 2 or 3 are more suitable. They can, however, be implemented by relaying incoming messages to other services. For space reasons, we omit this discussion.

[4] The functionality for declaring connections as reliable allows for resending messages when having temporary connection problems.

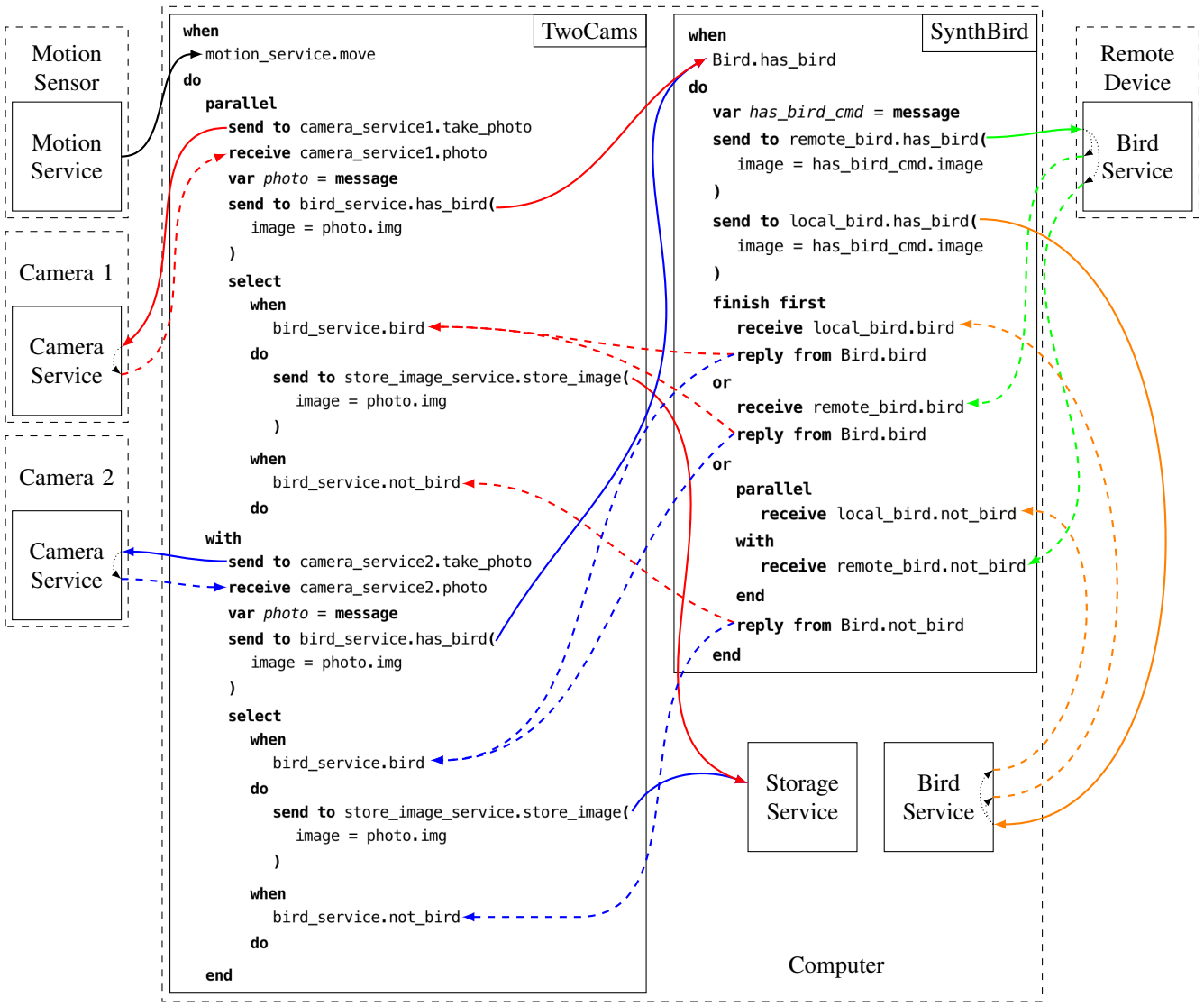

Fig. 4. Compositions used in the extended scenario. TwoCams (left) is a modified version of the original composition. SynthBird (right) is a composition of two different bird recognizers. An initial *move* message to TwoCams leads to two parallel subreactions (red and blue), leading to two corresponding reaction instances in SynthBird.

(4.0.19)[5]. Åkesson [11] created an experimental composition language for Palcom, optimized for latency-critical distributed applications. This language is similar to ComPOS in that compositions have state and support nested and parallel action sequences, but differs in the semantics of new spontaneous messages that arrive during a reaction. In Åkesson's approach, new messages start a parallel reaction (option 1 in section III-C), and indefinitely running reactions are avoided using timeouts. This is in contrast to ComPOS, where the current reaction is aborted (option 4). Another difference is that Åkesson's language is purely text-based, without any integration with a GUI, and is not intended for end users.

[5]http://palcom.cs.lth.se/Palcom/Download/Download.html

B. Web-service composition

Web-service composition has similarities to IoT service composition, but differs in that web services are assumed to be always available, wheras IoT services may come and go.

Examples of languages for web-service composition are Jolie [12] and BPEL [13]. These languages have similar features to ComPOS, with support for both parallel and finish first actions. A main difference is, however, that Jolie and BPEL support general computation rather than focusing on composition, and they target professional developers rather than end users like Maria from our example.

C. AmbientTalk

AmbientTalk [14] is a domain specific language developed for programming message-based applications in mobile ad-hoc networks. It is thus similar to ComPOS in its application

TABLE I
A UTILITY ANALYSIS OF THE SYSTEM SHOWN IN FIGURE 4.

Disconnected devices	Status	Reason
Computer	Useless	The system has nowhere to store images.
Motion sensor	Useless	No *move* messages arrive to start a reaction.
one Camera	Useful	The other camera can still take a photo and store it because the branch associated with that camera works as intended. For every new *move* message, the TwoCams composition creates a new activation and aborts the old one.
Remote Device	Useful	The synthesized service will never be able to send *not_bird*, but in the case the local bird service detects a bird the synthesized service will reply with *bird*.
both Camera	Useless	No camera to take the photo to be stored.
one Camera and Remote Device	Useful	If the local bird service detects a bird in a photo from the connected camera, that photo will be stored.
all other combinations	Useless	

domain, but differs in that it targets developers rather than end-users, and does not separate between oblivious services and compositions.

D. End-user development for IoT

There are different approaches for end-user development of IoT systems. Some use programming by demonstration [15], whereas others use different types of DSL:s, like TeC [16], Midgar [17], and AppsGate [18].

TeC [16] is a framework with the goal of allowing end users in different domains to create IoT applications. Similar to ComPOS, TeC has a distributed programming model with services (called *activities*) and compositions (called *team designs*). However, its computational model is quite different: activities have a kind of declarative spreadsheet semantics with input and output events, and can be adapted by the user. The team designs wire together input and output events of activities, but do not themselves contain any event logic or message adaptation.

Midgar [17] is a system that uses a graphical language to enable users to create compositions. The programs in Midgar are compiled and run on a central server.

AppsGate [18] is an end-user development environment, specifically intended for programming smart homes. Similar to ComPOS, the user uses a structure-oriented editor for programming the environment, but AppsGate uses a pseudo-natural language resembling English as its concrete syntax. AppsGate supports event rules similar to when-dos in ComPOS, but without any notion of request-responses, parallel actions, or synthesized services as in ComPOS, thus limiting the expressivity. AppsGate programs run on a central node in the network, and the program implicitly keeps track of the states of connected components, and supports relating them using *state rules*. An example of a state rule is "While temperature < 21 then keep the heater on". In contrast, ComPOS scripts can be executed on different nodes in the network, and all communication is based on explicit messages.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have presented ComPOS, a DSL for composing services into IoT systems. ComPOS is a new DSL supporting robust behavior in the presence of weak connectivity. We have introduced the notion of utility analysis and applied it to an example. We have shown how Maria, a fictive bird watcher, can use ComPOS to build a simple bird-watching system (Section II). To give an example of the abstraction mechanism (synthesized services) and the parallel construct, we extended the example to use two cameras and two bird recognizer services (Section IV). In an IoT system with mobile devices it is more a rule than an exception that devices disconnect from the network for one reason or another. To illustrate how ComPOS handles this, we have used our utility analysis to analyse what happens when devices in the system disconnect (Section V). In this particular example, we showed that up to two devices could fail and the system would still be useful. From the utility analysis, we conclude that it is possible to build systems using ComPOS that are useful even if some devices get disconnected.

In the future, we plan to build a tool that automatically analyzes how useful a system is with respect to connection failures. We would also like to continue looking into the end-user programming perspective of ComPOS, and do user studies to evaluate its usability.

ACKNOWLEDGMENT

This work was in part supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and in part by the Swedish Foundation for Strategic Research, grant RIT17-0035.

REFERENCES

[1] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller, "A survey of commercial frameworks for the internet of things," in *IEEE International Conference on Emerging Technologies and Factory Automation: 08/09/2015-11/09/2015*. IEEE Communications Society, 2015.

[2] S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang, "A vision of IoT: Applications, challenges, and opportunities with China perspective," *IEEE Internet of Things journal*, vol. 1, no. 4, pp. 349–359, 2014.

[3] R. Petrolo, V. Loscri, and N. Mitton, "Towards a smart city based on cloud of things," in *Proceedings of the 2014 ACM international workshop on Wireless and mobile technologies for smart cities*. ACM, 2014, pp. 61–66.

[4] A. Taivalsaari and T. Mikkonen, "A roadmap to the programmable world: software challenges in the IoT era," *IEEE Software*, no. 1, pp. 72–80, 2017.

[5] D. Svensson Fors, "Assemblies of pervasive services," Ph.D. dissertation, Department of Computer Science, Lund University, 2009.

[6] D. Svensson Fors, B. Magnusson, S. Gestegård Robertz, G. Hedin, and E. Nilsson-Nyman, "Ad-hoc composition of pervasive services in the PalCom architecture," in *Proceedings of the 2009 international conference on Pervasive services*. ACM, 2009, pp. 83–92.

[7] A. Åkesson, M. Nordahl, G. Hedin, and B. Magnusson, "Live programming of internet of things in Palcom," in *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. ACM, 2018, pp. 121–126.

[8] R. B. Smith, "Experiences with the alternate reality kit: An example of the tension between literalism and magic," in *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface*, ser. CHI '87. ACM, 1987, pp. 61–67.

[9] A. Åkesson, M. Nordahl, G. Hedin, and B. Magnusson, "Demo: A DSL for composing IoT systems," in *Proceedings of the 19th ACM/IFIP Middleware Conference: Posters and Demos*, 2018.

[10] A. Åkesson and G. Hedin, "Jatte: A tunable tree editor for integrated DSLs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*, ser. CoCoS 2017. ACM, 2018, pp. 7–12.

[11] L. Åkesson, *On the design of connector languages for latency-critical distributed applications.*, ser. Licentiate thesis 2016:1. Department of Computer Science, Lund University, 2016.

[12] F. Montesi *et al.*, "Service-oriented programming with Jolie," in *Web Services Foundations*, A. Bouguettaya *et al.*, Eds. New York, NY: Springer New York, 2014, pp. 81–107.

[13] "Web Services Business Process Execution Language Version 2.0," OASIS, Standard, Apr. 2007.

[14] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter, "AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks," in *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, Nov 2007, pp. 3–12.

[15] T. J.-J. Li, Y. Li, F. Chen, and B. A. Myers, "Programming IoT devices by demonstration using mobile apps," in *End-User Development*, S. Barbosa, P. Markopoulos, F. Paternò, S. Stumpf, and S. Valtolina, Eds. Cham: Springer International Publishing, 2017, pp. 3–17.

[16] J. P. Sousa, D. Keathley, M. Le, L. Pham, D. Ryan, S. Rohira, S. Tryon, and S. Williamson, "TeC: end-user development of software systems for smart spaces," *International Journal of Space-Based and Situated Computing*, vol. 1, no. 4, pp. 257–269, 2011.

[17] C. G. García *et al.*, "Midgar: Generation of heterogeneous objects interconnecting applications. a domain specific language proposal for internet of things scenarios," *Computer Networks*, vol. 64, pp. 143–158, 2014.

[18] J. Coutaz and J. L. Crowley, "A first-person experience with end-user development for smart homes," *IEEE Pervasive Computing*, vol. 15, no. 2, pp. 26–39, Apr 2016.