ComPOS - a development environment for composing internet-of-things services

Alfred Åkesson

Licentiate thesis, 2019

Department of Computer Science
Lund University

ISBN 978-91-7895-364-6 (printed version)
ISBN 978-91-7895-365-3 (electronic version)
ISSN 1652-4691
Licentiate Thesis 2, 2019
Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: `alfred.akesson@cs.lth.se`
Webpage: `http://cs.lth.se/alfred-akesson/`

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2019

Abstract

Internet-of-things (IoT) systems consist of spatially distributed devices with services. Compared to desktop applications, IoT systems are always running and need to deal with unresponsive devices and weak connectivity. In this thesis, we examine the following question: *How can we simplify the development of IoT systems?* We begin to answer this question by proposing a domain-specific language (DSL), called COMPOS, for composing IoT services. In the DSL, the user specifies the *reaction* to a message. The reaction can be programmed to request and receive responses in sequence and parallel. COMPOS can abort a running reaction when a new message arrives; this is to support unresponsive devices and weak connectivity. We demonstrate our language by creating a bird-spying system that takes photos of a garden and then stores the ones containing a bird. The COMPOS editor supports live programming for programming a running system. Programming in our DSL is divided into three phases: finding services (*explore*), composing services (*assemble*), and abstracting compositions as new services (*expose*). When developing a DSL, it takes effort specifying the syntax and semantics, building the editor, and integrating with the middleware. To reduce the effort needed to experiment with our DSL, we have created a tool called JATTE. The tool is a generic projectional editor that can be tuned for different languages using attribute grammars. We have integrated the editor built with the tool into an IoT development environment supporting discovery of devices and services.

Acknowledgements

I like first of all, to thank my main supervisor, Görel Hedin, for giving me this opportunity and for all the support. I also want to thank Görel for teaching me about research and academic writing and presenting. Also, I want to thank my co-supervisor Boris Magnusson for all our discussions and for taking me on as a research assistant. I want to thank my other co-supervisor, Niklas Fors, for his support.

Huge thanks to my co-author, travel buddy and demo operator, Mattias Nordahl. Björn Johnsson, I thank you, especially for insight about how PalCom is used. Thanks, Jesper Öqvist, my go-to guy for JastAdd problems. I like to thank the other members of the research SDE group Christoph Reichenbach, Noric Couderc, Alexandru Dura, and Emma Söderberg. Also, I like to thank all the members of the computer science department for all fika and support.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Therefore I want to thank the Knut and Alice Wallenberg Foundation. I also want to thank all my fellow batch-1-WASP-AS students and teachers and all other people in WASP.

I detta sista stycke, vill jag tacka mina föräldrar, Bengt och Anna-Karin Åkesson för allt deras stöd och rådgivning. Jag vill även tacka mina syskon Albin, Alma och Allis Åkesson samt mina far- och morföräldrar Åke och Elisabeth Andersson och Nils-Eric och Kerstin Andersson samt min övriga släkt för allt stöd. Jag vill tacka mina vänner som har givit mig inblick i "verkligheten". Ett tack även till alla i equmenia Nävlinge och Rickarum som har givit mig en meningsfull fritid och till alla mina syskon i Equmeniakyrkan Nävlinge-Rickarum för omsorg och förböner. Ett tack till den treeniga Guden för att ha, bland annat, skapat en värld där datorer existerar.

This thesis is a compilation consisting of an introduction, two papers, and a technical report. The technical report is a revisited and extended version of a paper.

Contributions of the author

List of included peer-reviewed publications by the thesis author.

Paper I Alfred Åkesson and Görel Hedin. "Jatte: A Tunable Tree Editor for Integrated DSLs". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*. CoCoS 2017. Vancouver, BC, Canada, 2018, pp. 7–12

The thesis author did all of the technical work and is the main author of sections 3-7.

Paper II Alfred Åkesson, Mattias Nordahl, Görel Hedin, and Boris Magnusson. "Live Programming of Internet of Things in PalCom". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Nice, France, 2018, pp. 121–126

The thesis author created the example and is the main author of sections 3-5.

Paper III *Extended version of:* Alfred Åkessson, Görel Hedin, Boris Magnusson, and Mattias Nordahl. "ComPOS: Composing Oblivious Services". In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. Kyoto, Japan, Mar. 2019, pp. 132–138

The thesis author did all of the technical work and is the main author of this paper as well as the extended version presented in this thesis.

Other publications

Alfred Åkesson, Mattias Nordahl, Görel Hedin, and Boris Magnusson. "Demo: A DSL for composing IoT systems". In: *Proceedings of the 19th ACM/IFIP Middleware Conference: Posters and Demos*. Rennes, France, 2018, pp. 17–18

Alfred Åkesson. "DSL for End-user Service Composition". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Nice, France, 2018, pp. 239–240

CONTENTS

1 Introduction

The cost of computers is decreasing, and low-cost computers are getting better connectivity. More devices get a connected computer embedded into them. These communicating devices enable new types of systems to emerge. We refer to this trend as the Internet-of-Things (IoT).

An *IoT system* is a system containing multiple connected devices. An example of IoT systems is in home care, where kidney-failure patients can weigh themselves at home and automatically get their weight sent to the hospital [JM16]. Another example of IoT systems is in home-automation, for example, having the colour of a light indicating the energy consumption of the home [CC16].

In the paper "A Roadmap to the Programmable World" [TM17], Taivalsaari and Mikkonen point out some challenges with programming IoT systems. Two challenges of particular interest for this thesis are:

- IoT systems have *weak connectivity* where devices may disconnect and reconnect to the rest of the system at any time.
- IoT systems are *always running*, even if single devices may shut down or disconnect.

Another challenge is to understand what happens in the system and to understand the different dependencies between devices [WGB99]. There is potentially a lot of useful IoT systems that we can build. To create these IoT systems faster, we can make it so simple to build IoT systems that even end-users can do it [Tet+15].

How can we simplify the development of IoT systems? That is the main question we strive to answer with this research. We take steps to answer this question by proposing a domain-specific language (DSL) (paper III), a development environment with support for live programming (paper II), and a tool for experimenting with the DSL and the development environment (paper I). Our proposed DSL is called COMPOS and is designed to handle weak connectivity (paper III). Our development environment supports live programming [Tan90; Tan13] to enable users to explore and evolve always-running IoT systems (paper II). To enable us to experiment with the DSL, we have created a meta-tool for generating editors called JATTE. JATTE generates editors with support for end-user-friendly features such as projectional editing and drag-and-drop (paper I).

The research we present in this thesis is built on prior work. COMPOS is built on top of the PalCom middleware toolkit [SF09]. JATTE and COMPOS are implemented using reference attribute grammars [Hed00] (in the JastAdd metacompiler [HM03]).

As a part of our research method, we create artefacts [KV15]. Using the artefacts, we can then implement scenarios to show our contributions. For the creation of the artefact, we use an iterative approach. First, we make a prototype and use it to implement a scenario. From the experience of implementing the scenario, we design the next prototype.

The rest of this chapter includes three sections of background (sections 2-4). Then follows a summary of the contributions of the thesis (section 5). Finally, we present conclusions and future work (section 6). The first background section describes IoT middleware and PalCom (section 2). The next background section describes domain-specific languages and their benefits, as well as projectional editing and reference attribute grammars (section 3). The third background section explains live programming (section 4).

2 IoT middleware

Middleware [Ber96] is an abstraction between an application and the underlying platform. It often abstracts the network in order to make it easier to program distributed applications. COMPOS is built upon the PalCom middleware toolkit. In this section, we will first describe PalCom and then look at different classifications of IoT middleware to classify PalCom.

2.1 PalCom

PalCom [SF09; SF+09] is a service-based middleware toolkit. It provides a protocol allowing *devices* to continuously discover each other as well as the *services* that each device currently run. Devices are uniquely identifiable running instances of the PalCom middleware, hosting a set of services. The unique identifiers of devices, called *device id*, are used for addressing messages. Services use asynchronous message passing to communicate. Before two services can communicate, a *connection* needs to be set up between them. There are three kinds of services: *native* services, *composition*[1] services and *synthesized* services. Native services perform computations and interact with the physical environment. A native service is normally *oblivious*, meaning that it does not set up connections to other services. It also does not know the identity of the service at the other end of a connection. Compositions compose oblivious services into systems by setting up connections as well as mediating and adapting messages. A composition service can also provide its own oblivious services called synthesized services. Synthesized services is an abstraction that composes the functionality of many oblivious services into one oblivious service.

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. Figure 2 shows an instance of the conceptual model with two devices, two native services and two compositions. Composition *C1* provides a synthesized service.

[1]Compositions are called *assemblies* in paper I and paper II

Figure 3 shows a sketch of an interactive tool called the *PalCom Browser* used for discovering services and creating compositions. On the left in the sketch is a view showing the discovered devices and services on the network, and the right is an editor for creating and editing compositions. In our work, we have integrated a new editor for COMPOS into the PalCom Browser.

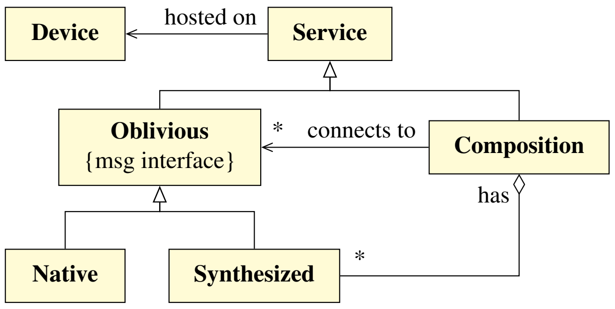


Figure 1: Conceptual model for PalCom devices and services.

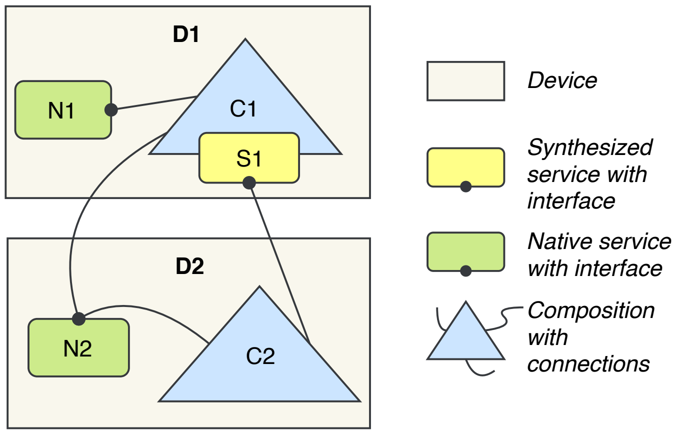

D1
N1
C1
S1
D2
N2
C2
Device
Synthesized service with interface
Native service with interface
Composition with connections

Figure 2: An instance of the conceptual model in figure 1.

2.2 Service-based, Cloud-based, or Actor-based?

There are many different middlewares supporting IoT systems. Ngu et al. [Ngu+17] propose to classify them into the following three categories:

Service-based A service-based middleware consists of services running either "in the cloud or on a powerful gateway" [Ngu+17]. There is no peer-to-peer

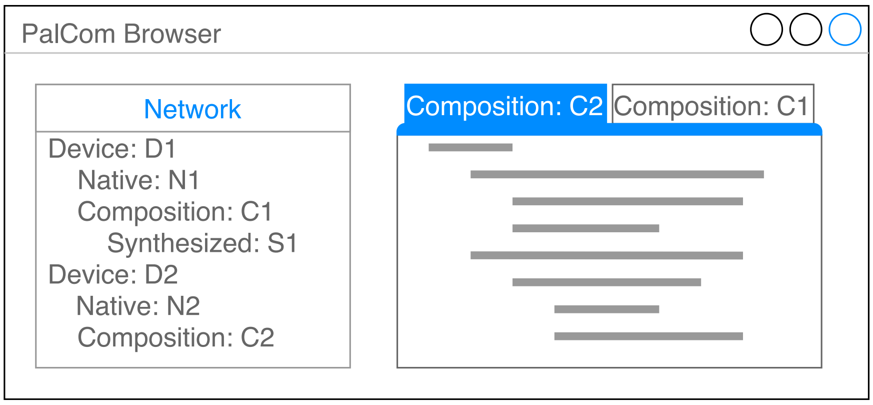


Figure 3: A sketch of the PalCom Browser with the discovered devices and services on the left and the composition editor on the right.

communication between the IoT devices; instead, the devices communicate with the services.

Cloud-based A cloud-based middleware is a vendor-provided service running in the cloud. Developers can only interact with the middleware through vendor applications or API:s.

Actor-based Actors are services that can be added dynamically to a device running the actor middleware. All devices in a system run the middleware. Every device that fulfils the hardware requirements of an actor can run it, e.g. a camera actor can run on every device with an image sensor. Actors allow for an open system where new devices can enter.

According to this classification, PalCom fits into the category of actor-based middlewares. PalCom services can be deployed on any device meeting the service's hardware requirements. By including a service in a composition, the service and the hosting device are added to the system, creating an open system.

We sometimes in this thesis call PalCom a service-based middleware. However, we do not think that PalCom fits the definition of service-based middleware given by Ngu et al. [Ngu+17]. When we say that PalCom is service-based, we allude to the fact that PalCom devices provide a set of services.

2.3 Communication Interaction Models

Eugster, Felber, Guerraoui, and Kermarrec [EFGK03] discuss different interaction models for distributed systems. They suggest three forms of decoupling between the sender and the receiver.

Space decoupling Interacting parties do not need to know the identity of each other to communicate.

Time decoupling Two interacting parties never need to be connected at the same time to exchange messages.

Synchronization decoupling The sending and receiving of messages happen outside the main flow of the program. The sender of a message does not need to block, waiting for a response, when sending a message. The receiver of a message gets asynchronously notified when a new message arrives and does not need actively to wait for messages to arrive.

	Space decoupling	Time decoupling	Synchronization decoupling
Message passing	No	No	Only sender
RPC	No	No	Only sender
Tuple space	Yes	Yes	Only sender
Pub/Sub	Yes	Yes	Yes
PalCom	Yes	No	Yes

Table 1: Different interaction models and their decoupling abilities. Grey parts from [EFGK03]. *Only sender* means the sending of message does not need to block, but the receiving of messages has to.

In table 1 [EFGK03] we see how PalCom compares to different interaction models. *Message passing* is a low-level form of interaction where the parties interact by sending messages to each other. *Remote Procedure Call (RPC)* makes the interaction with a remote machine look like a procedure call. RPC makes it easier to program distributed systems because there is no difference between calling a remote procedure or a local one. *Tuple space* [Gel85] is a set of tuples available to all participants in the interaction. Participants can put a tuple into the tuple set, pull a tuple out of the set and read a tuple from the set. In *Publish/Subscribe (Pub-/Sub)*, a publisher sends messages to a broker. A subscriber can then connect to the broker and subscribe to messages. A subscriber can define what messages it wants to subscribe to in many different ways. One way is that the publisher tags the message with a topic and the subscriber can then subscribe to that topic.

In PalCom, two oblivious services interact with each other through a composition. PalCom is space decoupled because oblivious services do not know to whom they talk. To have time decoupling, we need a third party to store the message when neither the sender or receiver is connected. A composition could act as such a third party, but this is not something we have implemented. Time decoupling could also be accomplished by having a caching service. In PalCom, messages are sent and received asynchronously, making PalCom synchronization decoupled.

2.4 Orchestration and Choreography

Two approaches for service composition are orchestration and choreography [Erl05]. In service orchestration, there is a central service sometimes called the conductor, which controls the messages sent between services. In choreography, the services act peer-to-peer, and coordinate messages among themselves without any central conductor. However, there is a global description describing different roles services can play in a choreography.

PalCom uses a combined approach with devices communicating peer-to-peer. Each composition acts as the conductor of a small orchestration. However, since compositions themselves may have synthesized services, the compositions constitute a graph distributed over different devices without a central conductor, corresponding to a choreography, without any description, at the macro level. In figure 2, we see each composition orchestrating services; at the same time, there is no central conductor in the system.

3 Domain-Specific Languages

In this section, we discuss domain-specific languages, projectional editing and reference attribute grammars.

A Domain-Specific Language (DSL) is a programming language designed for building applications in a specific domain. A DSL has constructs, notation and abstractions tailored for the domain [DKV00]. One of the contributions of this thesis is COMPOS, a DSL for composing IoT services.

Völter et al. [Völ+13] discuss a number of benefits and problems of DSLs. Below we discuss the benefits of particular interest for this thesis:

Productivity A COMPOS script would require writing less code than an equivalent program in a general-purpose programming language, and thus speeding up IoT system development.

Validation and Verification A script written in COMPOS contains much semantic information that can be used when designing analyses of IoT systems.

Productive Tooling Having COMPOS as a DSL allows us to create a custom editor using JATTE. We have integrated the editor into the PalCom Browser to support high-level domain-specific editing, e.g., allowing the user add a message send by dragging a message type from the "Network" window (figure 3) to the composition.

Below, we list some of the potential problems with DSLs identified by Völter et al. [Völ+13], and discuss some ways we try to tackle them:

Evolution and Maintenance When experimenting with the COMPOS language, it is often hard to have backward compatibility. One benefit of using projectional editing, compared to textual editing and parsing, is that changes in the concrete syntax are backwards compatible, as long as the abstract syntax is not changed.

Tool Lock-in COMPOS is built using JastAdd and JATTE, so replacing the underlying framework would be a considerable investment. COMPOS uses XML for serialisation, so creating a parser would be straightforward if COMPOS were to be reimplemented in another framework.

Learning It takes effort to learn COMPOS, but by using projectional editing, users do not have to learn the syntax.

Effort of Building the DSLs It takes effort to develop a DSL such as COMPOS. We use JATTE and JastAdd to speed up the development.

3.1 Projectional editing

When editing the source code of a program in an Integrated Development Environment (IDE), the IDE parses the code and builds an internal representation, typically in the form of an Abstract Syntax Tree (AST). The AST is used in analyses to provide feedback such as error messages and code completion. In *projectional editing* [VL14], also known as structural editing [Han71], instead of interacting with text, the user interacts with the AST [VL14]. The editing is done using operations for adding, removing, moving and changing AST-nodes. In COMPOS, the AST is visualised to the user using a textual notation. Our meta-tool JATTE, discussed in paper I, generates projectional editors from a reference attribute grammar.

3.2 Reference Attribute Grammars

An *attribute*, in attribute grammars [Knu68], is a property of an AST node, defined by a pure function that can access other attributes. There are two classes of attributes: synthesized and inherited. A synthesized attribute is declared on a node class and is like a virtual method. The defining function of a synthesized attribute is in the class or a subclass. An inherited attribute is also declared on a node class, but its defining function is in an ancestor node. Inherited attributes are useful for accessing information higher up in the AST, e.g., finding visible declarations.

Reference Attribute Grammars (RAGs) [Hed00] are attribute grammars where an attribute value can be a reference to another node in the AST. RAGs are useful for instance in name analysis, giving every use of a name a reference to its definition.

JastAdd

JastAdd [HM03] is the meta-compilation system we use to implement JATTE and COMPOS. In JastAdd the programmers specify their compilers using reference attribute grammars and aspect-oriented programming.

Aspect-oriented programming is a mechanism for modularisation of cross-cutting concerns [Kic+97]. JastAdd supports aspect-oriented programming by allowing different members of a class to be defined separately in different aspects, like open classes in MultiJava and inter type declarations in AspectJ [CLCM00; Kic+01]. These aspects can be used to separate different parts of a compiler implementation. Name analysis, type analysis and interpretation, are examples of different aspects. Aspects allow class members to be defined in aspect files, syntactically outside of their respective classes. In JastAdd, the members can be fields, methods, or attributes. The aspect files are then used by JastAdd to generate Java implementations of the AST classes. Figure 4 illustrates how JATTE, JastAdd, PalCom and COMPOS interact to generate the editor.

In JATTE, the user can customise the editor by overriding attributes. In JastAdd there are two ways of overriding attributes, either by overriding in a subclass like in Java or by specifying the aspect of the attribute to override.

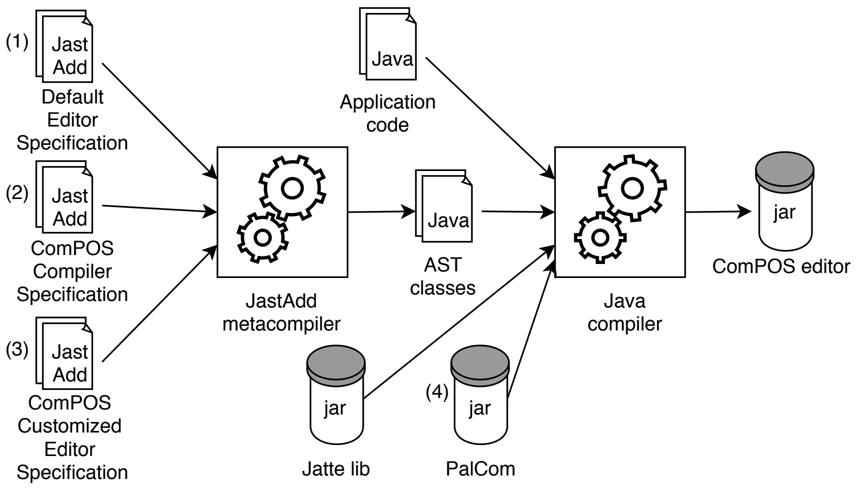


Figure 4: A sketch of how JATTE, JastAdd, PalCom and COMPOS interact to generate the editor for COMPOS The customized COMPOS editor (3) overrides attributes in the default JATTE editor (1), and can use attributes in the COMPOS compiler (2) and library functions in PalCom (4) to provide advanced editing support.

4 Live Programming

Live programming is about editing the program while it is running. The goal of live programming is to minimise the time from when the programmer edits the program until the programmer sees the result. The whole development environment is involved in supporting live programming. Paper II argues for using live programming when building IoT systems, addressing the *always running* challenge from section 1.

To classify how well a development environment supports live programming, Tanimoto has identified six levels of liveness [Tan13; Tan90]:

1. *Informative* To run the program, the user has to manually convert the program to a lower level language, for example, converting a class diagram to Java code.
2. *Significant* The user runs the program with a click of a button.
3. *Responsive* The development environment reruns the program after every edited operation. Useful for short running programs.
4. *Live* The development environment updates the running program after an edit. For example, changing colour in a game and see the result without restarting the game.
5. *Tactical predictive* The development environment tries to predict the next line the programmer will write and execute it. For example, the programmer opens a file, and the development environment automatically starts reading from it.
6. *Strategical predictive* The development environment tries to predict a large chunk of code. For example, automatically create a parser for the file the programmer just opened.

In paper II we argue that the PalCom-development environment supports liveness between level 3 and level 4.

5 Contributions

In this section, we describe the contributions of the individual papers and also the artefacts that were developed during this research.

5.1 JATTE: A Tunable Tree Editor for Integrated DSLs

In paper I, we present JATTE, a tunable projectional editor. Using JATTE, we show how reference attribute grammars can be used for specifying and tuning projectional editors. JATTE generates a default editor from the abstract grammar. The editor can then be tuned by overriding the default attributes JATTE generates. The tuning is used to specify the text, formatting, menu, and visibility of an AST node. In the paper, we build two editors using JATTE, one for a toy language and one for COMPOS. We also propose a way of integrating projectional editors into applications using JATTE. This is in contrast to most other projectional editors that come with their own interactive environment and are not intended to be integrated into other applications. One way we support integration is the support for drag-and-drop between the application and the editor. As an example of JATTE's support for editor integration, we integrated the COMPOS editor into the PalCom browser.

List of contributions in paper I:

- A new technique for developing projectional editors, based on reference attribute grammars.
- Examples showing how a generic projectional editor can be tuned to support context-sensitive editing, by overriding attributes.
- Examples showing how a projectional editor can be integrated into another application.
- Experimental validation by implementing the approach and applying it to two different languages, one of which is integrated into an existing application.

5.2 Live Programming of Internet of Things in PalCom

In paper II, we discuss using live programming to compose IoT systems in PalCom. Live programming allows the programmer to evolve running systems. We divided the PalCom live programming experience into three phases: explore, assemble and expose. The explore phase is about discovering and interacting with services, trying to understand their functionality in an exploratory way. The assemble phase is about composing services using our DSL. The expose phase is about exposing the functionality of a composition as a new service, i.e., creating a synthesized service. We show how the PalCom browser supports these three phases. In the paper, we also argue that PalCom supports liveness between level 3 and level 4 from Tanimoto's levels of liveness.

List of contributions in paper II:

- A characterisation of live IoT programming as a process of three interrelated phases: explore, assemble, and expose.
- A demonstration, showing this process in action.
- Arguing that PalCom supports liveness between levels 3 and 4.

5.3 COMPOS: Composing Oblivious Services

In paper III, we introduce COMPOS, a DSL for composing services in IoT systems. In the DSL, connections to services and devices are specified, making dependencies between devices explicit. A COMPOS script contains a list of guarded reactions. When a service spontaneously sends out a message that matches a guard, the composition starts to execute the reaction. The reaction contains actions for sending and receiving messages, expressing alternatives and doing nested actions in parallel. When a guard matches, it may abort an already running reaction triggered by the same service (see the paper for details about when this happens). Using weak connectivity as an argument, we motivate why we abort running reactions. For cases where other semantics than to abort is desired, we show how generic services can be used to adapt compositions semantics. We also propose *utility analysis*, an analysis used to determine what devices can fail, while still supporting partial functionality.

To demonstrate our contributions, we use a scenario with a system that helps a birdwatcher to spy on birds in a garden. We gradually extended the scenario with more devices to illustrate the features of the language. We also apply the utility analysis on the scenario.

List of contributions in paper III:

- A new DSL for composing IoT services with weak connectivity, including syntax and semantics in the form of an interpreter.
- A case study motivating the constructs in the language and their semantics.
- Examples showing how generic services can be used to change the semantics of compositions.
- The notion of a utility analysis to determine the usefulness of a system when different devices in it fail.

Developed artefacts

JATTE JATTE is a framework for creating projectional editors using RAGs and aspect-oriented programming. JATTE is built using JastAdd and uses Java Swing for rendering. Our implementation is open source available at `https://bitbucket.org/jastadd/jatte`, and an artefact evaluation is at `https://bitbucket.org/jastadd/jatteartifactevaluation/downloads/`.

COMPOS COMPOS consists of two parts, an editor and an interpreter. The interpreter is implemented in JastAdd and uses the PalCom middleware for all its communication. The editor is implemented using JATTE and integrated with the PalCom browser. Videos demonstrating COMPOS are available at `https://lu.box.com/s/wxc9y5psxfk91li4027r88crd1tbe1yj`.

6 Conclusions and Future work

In this thesis, we explore ways to simplify the development of IoT systems. IoT systems are hard to understand with nontrivial dependencies between devices. Our approach is to design a DSL, called COMPOS, to make the connections in the system explicit and thereby making the dependencies easier to understand. IoT systems are weakly connected; hence, COMPOS is designed to handle connections going down. We also explored how to evolve always-running IoT systems with live programming. Implementing a DSL with editor support takes effort. To allow us to experiment with our DSL efficiently, we created JATTE, a tool for creating projectional editors and integrating them in applications.

In the future, we see three main lines of continued research. The first is to look into what we can analyse to get an understanding of a connected system. The first step to that end will be to try to implement the utility analysis described in paper III. Then we can look into other analyses. One idea is adding more expressive interface descriptions to services that allow us to analyse how messages flow through a system. Once we can analyse a whole system, we can generate overviews similar to figure 2. The second line of research is to evaluate COMPOS more extensively by conducting a larger case study by reimplementing the compositions in a system for home care with COMPOS. The third line of research is looking at the usability aspects of the development environment. Ideally, we would like to support that end users, without programming experience, can compose IoT systems using COMPOS. The user studies will hopefully give indications of how usable the development environment is, and insight into how we can improve usability [NM90].

Further, in the future, we may be able to reach liveness level 5 and 6 by leveraging opportunistic composition engines [YTAA18].

References

[Ber96] Philip A. Bernstein. “Middleware: A Model for Distributed System Services”. In: *Commun. ACM* 39.2 (Feb. 1996), pp. 86–98.

[CLCM00] Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. “MultiJava: Modular open classes and symmetric multiple dispatch for Java”. In: *ACM Sigplan Notices*. Vol. 35. 10. ACM. 2000, pp. 130–145.

[CC16] J. Coutaz and J. L. Crowley. “A First-Person Experience with End-User Development for Smart Homes”. In: *IEEE Pervasive Computing* 15.2 (2016), pp. 26–39.

[DKV00] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography”. In: *SIGPLAN Notices* 35.6 (2000), pp. 26–36.

[Erl05] Thomas Erl. “Service-Oriented Architecture: Concepts, Technology, and Design”. In: Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. Chap. 6.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. “The Many Faces of Publish/Subscribe”. In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131.

[Gel85] David Gelernter. “Generative Communication in Linda”. In: *ACM Trans. Program. Lang. Syst.* 7.1 (Jan. 1985), pp. 80–112.

[Han71] Wilfred J. Hansen. “User engineering principles for interactive systems”. In: *AFIPS ’71 Fall Joint Computer Conference*. ACM, 1971, pp. 523–532.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[HM03] Görel Hedin and Eva Magnusson. “JastAdd–an aspect-oriented compiler construction system”. In: *Sci. of Comp. Prog.* 47.1 (2003), pp. 37–58.

[JM16] Björn A Johnsson and Boris Magnusson. “Supporting collaborative healthcare using PalCom–The itACiH system”. In: *Pervasive Computing and Communication Workshops (PerCom Workshops), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–6.

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-oriented programming”. In: *ECOOP’97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[Kic+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. “An overview of AspectJ”. In: *ECOOP*. Vol. 2072. LNCS. Springer. 2001, pp. 327–354.

[Knu68] Donald E. Knuth. “Semantics of Context-free Languages”. In: *Math. Sys. Theory* 2.2 (1968). Correction: Math. Sys. Theory 5(1):95–96, 1971, pp. 127–145.

[KV15] Shriram Krishnamurthi and Jan Vitek. “The Real Software Crisis: Repeatability As a Core Value”. In: *Commun. ACM* 58.3 (Feb. 2015), pp. 34–36.

[Ngu+17] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng. “IoT Middleware: A Survey on Issues and Enabling Technologies”. In: *IEEE Internet of Things Journal* 4.1 (2017), pp. 1–20.

[NM90] Jakob Nielsen and Rolf Molich. “Heuristic Evaluation of User Interfaces”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’90. ACM, 1990, pp. 249–256.

[SF09] David Svensson Fors. “Assemblies of pervasive services”. PhD thesis. Department of Computer Science, Lund University, 2009.

[SF+09] David Svensson Fors, Boris Magnusson, Sven Gestegård Robertz, Görel Hedin, and Emma 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.

[TM17] Antero Taivalsaari and Tommi Mikkonen. “A roadmap to the programmable world: software challenges in the IoT era”. In: *IEEE Software* 1 (2017), pp. 72–80.

[Tan90] Steven L. Tanimoto. “VIVA: A visual language for image processing”. In: *Journal of Visual Languages & Computing* 1.2 (1990), pp. 127 –139.

[Tan13] Steven L Tanimoto. “A perspective on the evolution of live programming”. In: *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press. 2013, pp. 31–34.

[Tet+15] Daniel Tetteroo, Panos Markopoulos, Stefano Valtolina, Fabio Paternò, Volkmar Pipek, and Margaret Burnett. “End-User Development in the Internet of Things Era”. In: *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’15. Seoul, Republic of Korea: ACM, 2015, pp. 2405–2408.

[VL14] Markus Völter and Sascha Lisson. “Supporting Diverse Notations in MPS’ Projectional Editor.” In: *GEMOC@MoDELS*. 2014, pp. 7–16.

[Völ+13] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. “DSL Engineering - Designing, Implementing and Using Domain-Specific Languages”. In: dslbook.org, 2013. Chap. 2, pp. 40–43,71,78.

[WGB99] M. Weiser, R. Gold, and J. S. Brown. “The origins of ubiquitous computing research at PARC in the late 1980s”. In: *IBM Systems Journal* 38.4 (1999), pp. 693–696.

[YTAA18] Walid Younes, Sylvie Trouilhet, Françoise Adreit, and Jean-Paul Arcangeli. “Towards an Intelligent User-Oriented Middleware for Opportunistic Composition of Services in Ambient Spaces”. In: *Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things*. M4IoT’18. ACM, 2018, pp. 25–30.

PAPER I

JATTE: A Tunable Tree Editor for Integrated DSLs

Abstract

Complex systems often integrate domain-specific languages to let users customize the behavior. Developing tooling for such languages is typically time-consuming and error-prone. We present JATTE, a tool intended to simplify this development. JATTE works as a generic tree editor for an abstract syntax, but uses aspects and attribute grammars to support powerful modular ways of tuning both the projected view and the editing commands. We present the key features of JATTE, and discuss its application in an orchestration language for internet of things.

1 Introduction

Complex systems often integrate domain-specific languages (DSLs) to let users customize the behavior. To support close and comprehensible integration with the application, structural/projectional editing [Han71; TR81; Rei85; Dmi04; Völ09] can be preferable to textual editing. In particular, parts of the DSL program may refer to entities in the system that are difficult to understand as text, and which can be suppressed by a projectional editor. For example, in an orchestration DSL for an Internet-of-Things (IoT) system, the globally unique name of a particular device is typically a long string, incomprehensible to a human. A projectional editor can instead show a human readable name.

Alfred Åkesson and Görel Hedin. “Jatte: A Tunable Tree Editor for Integrated DSLs”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*. CoCoS 2017. Vancouver, BC, Canada, 2018, pp. 7–12

Furthermore, DSL users are typically not expert programmers, and it is useful for the integrated editor to have intelligent editing support, like drag-and-drop between visual representations of the system parts and the DSL program, as well as context-dependent support like code completion.

Generic structural tree editors can be a good starting point for constructing integrated DSL editors. One example is the Eclipse Modelling Framework tree editor, *EMF.Edit* [SBMP08]. However, customizing such editors can be difficult. There is an example in the EMF book on how to hide a node type in *EMF.Edit*, requiring changing the generated Java code, adding 61 non-trivial lines of code.

Customization needs to be done in a much easier way, and without having to resort to fragile practices like changing generated code. To support these needs, we have implemented a general tool, JATTE, that supports easy and powerful customizable tree editing, where customization is added using modular aspects. JATTE can be tuned to customize what AST nodes are shown, how they are shown, and what edit commands are provided. It also supports customized editing like drag and drop from other parts of an application into which the editor is integrated. We believe such customization and close integration with the application is often vital to the comprehensibility of the DSL.

The customization is done using reference attribute grammars [Hed00] as supported by the JastAdd metacompilation tool [HM03]. We show through examples how this gives a powerful way of customizing the editor, easily supporting context-dependent facilities like intelligent code completion.

We start by giving some brief background on JastAdd and attribute grammars (Section 2). We then present how JATTE works out of the box as a generic tree editor (Section 3). In Section 4 we present our main contribution: how the editor can be customized using attribute grammar aspects.[1] We then present a case study where JATTE is used to implement an editor for an orchestration language for the IoT middleware PalCom [SF+09] (Section 5). Finally, we briefly discuss the implementation of JATTE (Section 6), related work (Section 7), and conclude (Section 8).

2 Background

JastAdd is a compiler construction system supporting aspects and attribute grammars. In JastAdd, the developer specifies an abstract grammar that is equivalent to a Java class hierarchy. A clause like

$Class : SuperClass ::= Right\text{-}hand\text{-}side;$

specifies an AST node class, where the right-hand side declares individual children, list children, optional children, and tokens. The developer can then add methods, attributes, and equations to the node classes. This is done modularly us-

[1] JATTE is open source. The tool and a video can be downloaded at `https://bitbucket.org/jastadd/jatteartifactevaluation/downloads/`.

ing aspects with inter-type declarations, like in MultiJava and AspectJ [CLCM00; Kic+01].

An *attribute* is a derived property of an AST node, defined by a directed equation whose right-hand side is a function that may access other attributes in the AST.

Attributes are classified as synthesized or inherited. A *synthesized* attribute is declared on a node class and is similar to a virtual function, with its defining equation in the class or a subclass. An *inherited* attribute is also declared on a node class, but its defining equation is located in an ancestor node. Inherited attributes are useful for accessing information higher up in the AST, like visible declarations [HM03; Knu68]. The JastAdd tool weaves attributes and methods from the aspect files into Java classes generated from the abstract grammar.

3 Default Tree Editor

By default, the JATTE tree editor displays each AST node as a row, and the node's children as nested rows. The displayed label for a node is derived from its name (as seen from the parent), its actual type, and any tokens with their values:

$Name : ActualType < Token = TokenValue >$

The abstract grammar in Fig. 1 shows a tiny calculator language, *Calc*, with `Let` expressions. Fig. 2 shows the corresponding default editor. The user has added a *Mul* node with two children of type `Numeral`, one with the token value 1, and one with the value 2. Menus for editing the tree are generated based on the abstract grammar, an example of this is shown in Fig. 3. A node can be replaced by a node of another type, if the change follows the grammar, and tokens can be edited as text. For lists and optionals, there are additional generic commands to add and remove nodes.

When adding a node, the AST is automatically completed to a full subtree. For example, when adding a `Mul` node, its two operand children will be added as well. Heuristics are used to construct a subtree with few children and tokens.

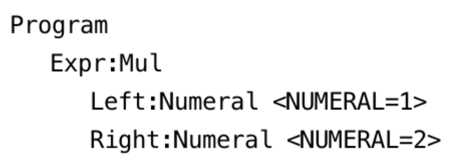
The editor supports saving the AST, serializing to it XML.

4 Customizing the Editor

The default editor can be customized by, for example, changing the node labels, hiding nodes, and changing the menu. The editor behavior is controlled by a number of attributes declared on the class `ASTNode`, which is an implicit superclass of all node classes. For example, there is an attribute `ed_label` for specifying the label of a node. The user can add aspect files with equations that override these attributes for specific node classes. By introducing helper attributes, synthesized or inherited, powerful customization is supported, as will be illustrated.

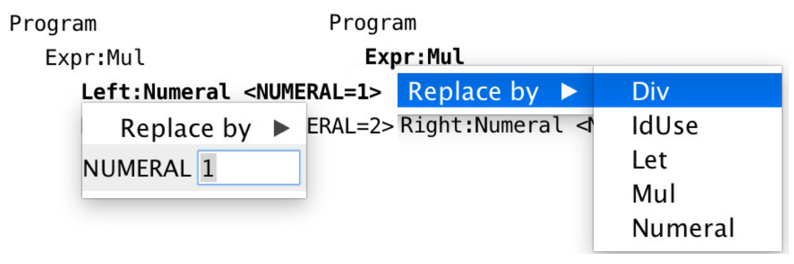
```
Program ::= Expr;
abstract Expr;
Mul : Expr ::= Left:Expr Right:Expr;
Div : Expr ::= Left:Expr Right:Expr;
Numeral : Expr ::= <NUMERAL>;

Let : Expr ::= Binding* Expr;
Binding ::= IdDecl Expr;
IdDecl ::= <ID>;
IdUse : Expr ::= <ID>;
```

Figure 1: Abstract grammar for tiny *Calc* language



```
Program
   Expr:Mul
      Left:Numeral <NUMERAL=1>
      Right:Numeral <NUMERAL=2>
```

Figure 2: Default editor for the language in Fig. 1



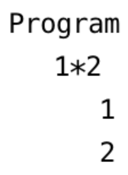
```
Program
   Expr:Mul
      Left:Numeral <NUMERAL=1>
```

```
Program
   Expr:Mul
      ERAL=2> Right:Numeral <N
```

Figure 3: Default menus for `Mul` and `Numeral` nodes.

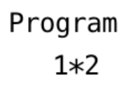
```
eq Expr.ed_label() = pp();
syn String Expr.pp() = "";
eq Mul.pp() =
   getLeft().pp() + "*" + getRight().pp();
eq Div.pp() =
   getLeft().pp() + "/" + getRight().pp();
eq Numeral.pp() = getNUMERAL();
```

Figure 4: Customizing the labels of expressions



```
Program
   1*2
      1
      2
```

Figure 5: Customized labels for `Mul` and `Numeral`



```
Program
   1*2
```

Figure 6: The `Numeral` nodes have been hidden.

4.1 Customizing Node Labels

The editor displays nodes by using the attribute `ed_label` of type `String`. Suppose we would like to change the default labels in *Calc* so that each expression is shown as a complete textual version. For example, we would like the node labelled `Expr:Mul` to instead be labelled `1*2`. This can be done by introducing a synthesized attribute `pp` for prettyprinting of expressions, and redefining `ed_label` using `pp`, as shown in Fig. 4. The result of this customization can be seen in Fig. 5. With slightly different attribution rules, we can easily define fully or minimally parenthesized expressions.

4.2 Hiding Nodes

It is often the case that some nodes are uninteresting to view in the editor. For example, we might like to hide the `Numeral` nodes that are children to `Mul` and `Div`-expressions. The editor uses a boolean attribute `ed_show` to determine if a node should be shown or hidden. If the value of the attribute is evaluated to `true` (default), the node is shown, else it is hidden. Fig. 6 shows what the tree looks like when `Numeral` nodes are hidden, using the equation:

```
eq Numeral.ed_show() = false;
```

When nodes are hidden, their menus are automatically merged into the menu of the closest visible ancestor. Fig. 7 shows the default generated menu for the program with hidden `Numeral` nodes. Here, the left and right `numerals` can be edited or replaced using the menu on the `Mul` node.

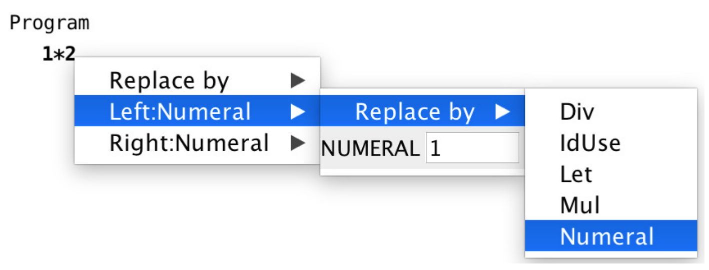


Figure 7: The menus for the hidden `Numeral` nodes have been merged into the `Mul` menu.

```
aspect show{
    eq Numeral.ed_show() = !parentHidesMe();
    eq IdUse.ed_show() = !parentHidesMe();
    inh boolean Expr.parentHidesMe();
    eq ASTNode.getChild().parentHidesMe() = true;
    eq Let.getExpr().parentHidesMe() = false;

    eq IdDecl.ed_show() = false;
    eq Binding.getChild().parentHidesMe() = true;
}
```

Figure 8: Context-dependent hiding of `Numerals`

Context-dependent Hiding. The hiding of nodes can be made conditional and context-dependent by defining `ed_show` using other attributes. As an example, consider the `Let` construct in Fig.1, which lets us bind a set of variables to expressions, and use these variables in the last expression. We would like the last expression to be shown in the editor, regardless of if it is a `Numeral` or not, but hide `Numeral` otherwise. We thus need the equation for `Numeral.ed_show()` to depend on the place where it is located in the tree. The `Numeral` should be shown when it is the `Expr`-child of a `Let`-node, but hidden otherwise. This can be accomplished by introducing an inherited attribute `parentHidesMe` for expressions. Since the attribute is inherited, its value is defined by an equation in an ancestor. We give a default equation for `parentHidesMe` in `ASTNode` (the superclass of all nodes in the tree), defining it to be `true` for all children, and then letting `Let` override this equation for its `Expr`-child, defining it as `false`. The `ed_show` attribute for a `Numeral` can then be defined using its `parentHidesMe` attribute. These equations are shown in Fig. 8. Fig. 9 shows a Let construct where the node for 10 inside the binding `a = 10` is hidden, but the node for 5 is shown.

```
Program
  Let
    a = 10
    5
```

Figure 9: The node for 10 is hidden, but 5 is visible.

```
abstract Menu ::= <name>; // Name displayed in menu
MenuList:Menu ::= Menu*;
MenuItem:Menu ::= <creator:ASTNode>; // node to act on
ReplaceType:MenuItem ::= <type:Class>;
...
```

Figure 10: Subset of abstract grammar for menus

4.3 Customizing Menus

The menu for a node is derived from a set of attributes, making it suitable for customization. Every node has an attribute `ed_menu` that represents its menu. This attribute is a *higher-order attribute*, i.e., its value is a fresh AST that can itself have attributes [VSK89]. This way, the structure of the menu is represented using an AST.

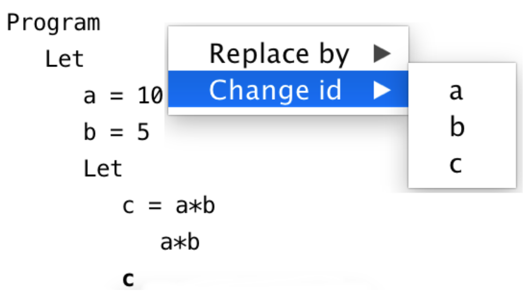
Fig. 10 shows a subset of the abstract grammar for menus. A menu is either a list containing sub-menus, or a menu item that can be selected. Each menu item has a method `perform` that implements what to do when the item is selected. Different subtypes have different behavior. For example, a `ReplaceType` menu item will replace the current node by a node of another type.

The default menu contains items depending on the node-type and on its location in the AST. For example, if the node is contained in a list or optional, there will be a menu item for removing the node. If the node has String tokens, there will be menu items to edit them. There are menu alternatives that enables the user to create all trees following the abstract grammar.

If a visible node has hidden children, their menu items are merged (recursively if needed) into the menu of the node. To handle menu items for hidden children, a reference to the node to act on is stored in the menu item's token `creator`.

The user can customize menus in several ways. Predefined attributes can be overridden to add new menu items, or hide default ones. When defining new menu items, the existing menu classes can be used, but the user can also define new subclasses and override the `perform` method to add custom behavior. It is also possible to override the `ed_menu` definition in order to define a completely different menu.

Intelligent Code Completion Menus. Through the use of attribute grammars, it is easy to add customizations for intelligent code completion. For example,



```
Program
  Let
    a = 10
    b = 5
    Let
      c = a*b
        a*b
      c
```

Figure 11: Intelligent code completion

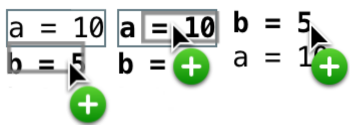


Figure 12: Using drag and drop to reorder a list

attributes can be added to support a menu of visible names when editing variables. Fig. 11 shows an example of this for the *Calc* language, where the names in the enclosing `Let` expressions are selectable from a menu. This functionality was implemented using 48 lines of attribute code.

4.4 Drag and Drop

JATTE supports Drag and Drop (DnD), allowing the user to drag information both between nodes inside the editor and between the editor and a surrounding application. All AST nodes have the attributes `ed_can_drag()`, returning *true* if the node can be dragged, and `ed_can_drop(Object resource)`, returning *true* if the `resource` can be dropped on the node. To define what happens when a drop is done, the method `ed_accept_resource(Object resource)` is used.

By default, these attributes are defined to allow elements in lists to be reordered using DnD, as shown in Fig. 12. Here, the `ed_can_drag()` is defined as *true* for all elements in the list. `ed_can_drop(Object resource)` is defined as *true* for elements in the same list as `resource`. Finally, `ed_accept_resource(Object resource)` is defined to move the `resource` node to the drop target node.

To make use of customized DnD, an aspect can be added that overrides or refines the definitions of these attributes and methods for particular node types. By using helper attributes, context-dependent DnD can be defined, for example, only allowing a node to be dropped at a type-correct location.

5 Case Study: IoT Language

The primary motivation for developing JATTE was the need for fast prototyping of a new version of an orchestration DSL for IoT. The language is called As2 (Assembly script 2), and is used in an IoT middleware called PalCom [SF+09]. An As2 script is used for connecting services on devices in a network, and mediating commands between them. Every service defines a set of in-commands that it can receive and a set of out-commands that it can send. The user can edit a script by dragging in- and out-commands from a panel of discovered services into the script's event handler. Structure editing and code completion can be used for editing details in the script.

5.1 A Scenario

As a simple example scenario, we will show how to construct an As2 script to connect a camera service with a database server that can store photos. Each time the camera snaps a photo, it should be sent over the network to the database where it is stored for later use.

In Fig. 13 we see a screenshot from an application called the *PalCom browser*. This application is used for discovering what devices and services are available on the network. The JATTE-based As2 editor has been integrated into the browser application. On the left in Fig. 13 we see a discovery panel with devices and services found on the network. There is a device called *Camera* with a service called *PictureService*. The service has an out-command called *picture*, and it is sent to any connected service every time the camera snaps a photo. There is also a device called *Server* with a service named *ImageDB* (Image database), with an in-command *storeImage*.

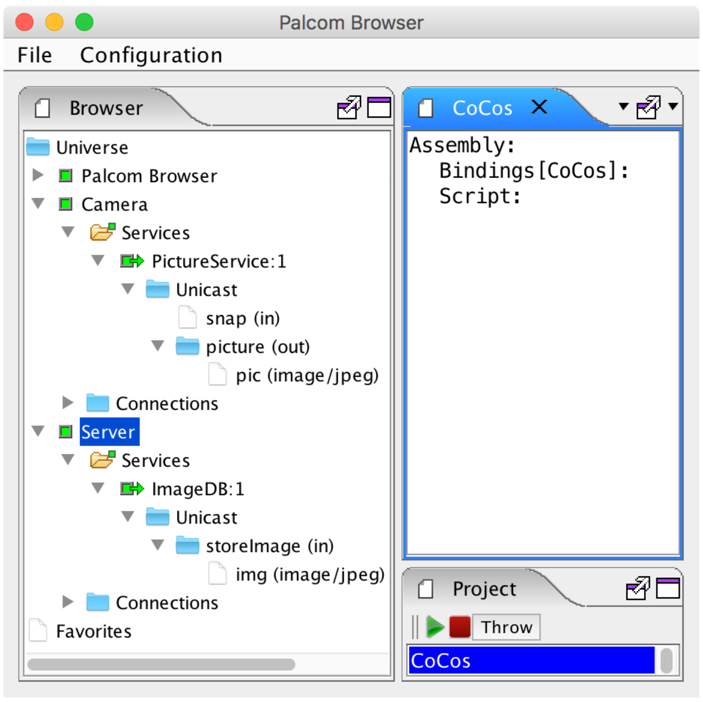
By using As2, we can create a script that connects to the two services and forwards the pictures from the camera to the database server.

We first create a new As2 file; this gives us the initial script shown to the right in Fig. 13. To create an event-handling case, we add a *when do* construct by selecting in the context menu for the *Script* node, see Fig. 14.

We then drag the out-command *picture* from the discovery panel and drop it on the *when* node, giving the script shown in Fig. 15. The event handler will react every time it receives a *picture* command from the *PictureService*.

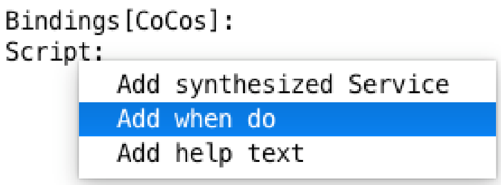
In addition to the *picture* command, the DnD command automatically adds local declarations of the device and service under the *Bindings* heading. For instance, the *Camera* device is bound to the local name *d0*. The DnD command also defines a local variable *local1* in the *when* clause, capturing the value of the image.

The next step in the As2-script is to forward the image to the *Server*. We do this by adding a *send to service*-action and then dragging the *storeImage* command to that action. This drop adds the service and device to *Bindings* the same way as for the *picture*-command. The parameter *img* in the forwarded message gets its



```
Assembly:
   Bindings[CoCos]:
   Script:
```

Figure 13: The PalCom browser with a discovery panel to the left and an As2 script to the right



```
Bindings[CoCos]:
Script:
```

Figure 14: The menu for adding *when do*

```
Assembly:
   Bindings[CoCos]:
      Device: d0 <- Camera
      Service: PictureService0 <- PictureService[1] on d0
   Script:
      when
         picture from PictureService0
            local1 <- pic
```

Figure 15: The script after dragging the *picture*-command

```
Assembly:
  Bindings[CoCos]:
    Device: d0 <- Camera
    Device: d1 <- Server
    Service: PictureService0 <- PictureService[1] on d0
    Service: ImageDB0 <- ImageDB[1] on d1
  Script:
    when
      picture from PictureService0
        local1 <- pic
      send storeImage to ImageDB0
        img <- local1
```

Figure 16: The final script

value from *local1*. Now we have completed our task and can save and run the script. In Fig 16 we see the full script.

5.2 The use of JATTE

We will now discuss how different features of JATTE are used to construct the As2 editor.

Customizing Node Labels. As an example of how we use the feature for customizing labels, we can look at the first row of *Bindings* where we bind a device to *d0*. The abstract grammar for this row is: `DeviceDef ::= DeviceParam:NameDef <deviceRefPON>;` Where `<deviceRefPON>` is a string containing structured information about the device encoded in the PON-format [NM16]. The user does not see this string; instead, we have defined the `ed_label` attribute to extract the readable name of the device from the PON-encoded string and show it to the user. Every device also has a UUID used by the interpreter for identification of devices. Both the UUID and the readable name are contained in the PON string and saved in the XML file. Since the UUID is irrelevant to the user, we think hiding it leads to better comprehension of the language.

Hiding Nodes. We use the feature for hiding nodes. In the abstract grammar, we have a node called *Definitions* containing the list of *ServiceDef* and the list of *DeviceDef*. This node is hidden from the user, and instead, the services and devices are visible directly under *Bindings*.

Customized Menus. The As2 editor customizes the menus for the nodes. The menu in Fig 14 has similar actions as the default generated menu but the actions are reordered and renamed with the use of attributes. This is done in an attempt to make the DSL more comprehensible and easier to use.

We have also implemented context-dependent code completion. In the abstract grammar of As2 we have a production called *NameUse*. *NameUse* is used every time we use an identifier. The *NameUse* can both refer to script-local identifiers and to things on the PalCom network. We have implemented code completion for *NameUse*, presenting visible script-local identifiers in a menu, using a similar method as described in Section 4.3. The visible names are defined in the ancestor nodes and different *NameUse* nodes therefore get different menus depending on their context.

Drag and Drop. JATTE's support for drag and drop is utilized in the As2 editor to interact with the surrounding application. The dragging of a service command eliminates the need for the user to ensure that the correct service and device is used for that command because all that information is inferred by the DnD action.

6 Implementation

JATTE is implemented using reflection. It makes use of annotations in the generated Java classes to find out about the actual abstract syntax, and communicates with the AST by calling the predefined attributes, for example `ed_label`. Attributes are evaluated on demand, i.e., when they are used. To avoid unnecessary recomputations during evaluation, the values can be automatically cached. However, whenever the AST is edited, the cached values could become inconsistent. Therefore, JATTE clears all caches in the whole AST after each edit operation. The user interface is then updated using the new values of the attributes. Because of the on-demand evaluation, the editor is fast enough for interactive use.

7 Related Work

JATTE is similar to EMF.EDIT in that both are general AST editors. In EMF, the AST is called a *model*, and containment references correspond to the tree structure. EMF.EDIT can also be customized, but this is done by changing generated Java code, and appears to be much more complex than in JATTE. For example, in chapter 19 of the EMF book [SBMP08], there is an example of how to hide nodes of type `USAddress` in a DSL for purchase orders, and delegate the properties of the hidden node to a parent node so they can be edited. To implement this in EMF.EDIT requires 61 lines of code. As a comparison, we implemented a similar example in JATTE, and the corresponding hiding and delegation was accomplished with the following single line of code:

```
    eq USAddress.ed_show() = false;
```

EMF.EDIT is, however, a mature tool, widely used, whereas JATTE is still a research prototype.

There are many text and structure-based editors that have used attribute grammars, starting with the Synthesizer Generator [RT84]. Our work is different in that we target general tree editors for languages with an abstract grammar only, and no concrete parsing grammar.

Language workbenches support development of advanced editing support for DSLs [Erd+15], but typically generate plugins to development platforms like Eclipse, and are therefore difficult to integrate tightly with an arbitrary application.

EuGENia is a framework for building diagram based editors on top of the Eclipse Graphical Modeling Framework (GMF), with the goal of having a higher abstraction level than GMF. The developer specifies an editor by adding annotations to the classes in the meta-model [KRPP09], somewhat similar to our adding of attributes to classes. However, attributes also support general computations, which allows more open ended customization.

8 Conclusion

Our work addresses the problem of how to easily develop powerful DSL editors, integrated into complex systems. To this end, we have demonstrated how customization is done in our tool JATTE, by overriding default behavior using attribute grammar equations. Examples include customization of displayed node labels, hiding nodes, customizing menus, and supporting drag and drop editing. Arguably, this technique is both easy to use and powerful, supporting advanced customizations like context-dependent node hiding and intelligent code completion with only a few lines of code. We have exemplified the use of JATTE for an orchestrating DSL in a research project on IoT, showing examples of advanced customizations like drag and drop from the application, and intelligent code completion. JATTE is still a research prototype, and is actively being improved. Because of the ease with which customizations can be added, JATTE is suitable for rapid prototyping of DSLs. An interesting direction of further research is to support grammar evolution, i.e., to allow existing programs following an older grammar version to be read in by the tool and adapted to the new grammar version.

Acknowledgments

This work was partially supported by the Wallenberg Autonomous Systems and Software Program (WASP). We thank Niklas Fors and the anonymous reviewers for helpful comments on earlier drafts of the paper.

References

[CLCM00] Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. “MultiJava: Modular open classes and symmetric multiple dispatch for Java”. In: *ACM Sigplan Notices*. Vol. 35. 10. ACM. 2000, pp. 130–145.

[Dmi04] Sergey Dmitriev. “Language oriented programming: The next programming paradigm”. In: *JetBrains onBoard* 1.2 (2004), pp. 1–13.

[Erd+15] Sebastian Erdweg et al. “Evaluating and comparing language workbenches: Existing results and benchmarks for the future”. In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47.

[Han71] Wilfred J. Hansen. “User engineering principles for interactive systems”. In: *AFIPS ’71 Fall Joint Computer Conference*. ACM, 1971, pp. 523–532.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000), pp. 301–317.

[HM03] Görel Hedin and Eva Magnusson. “JastAdd–an aspect-oriented compiler construction system”. In: *Sci. of Comp. Prog.* 47.1 (2003), pp. 37–58.

[Kic+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. “An overview of AspectJ”. In: *ECOOP*. Vol. 2072. LNCS. Springer. 2001, pp. 327–354.

[Knu68] Donald E. Knuth. “Semantics of Context-free Languages”. In: *Math. Sys. Theory* 2.2 (1968). Correction: *Math. Sys. Theory* 5(1):95–96, 1971, pp. 127–145.

[KRPP09] Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Fiona AC Polack. “Raising the level of abstraction in the development of GMF-based graphical model editors”. In: *MiSE@ICSE*. IEEE. 2009, pp. 13–19.

[NM16] Mattias Nordahl and Boris Magnusson. “A lightweight data interchange format for internet of things with applications in the PalCom middleware framework”. In: *Journal of Ambient Intelligence and Humanized Computing* 7.4 (2016), pp. 523–532.

[Rei85] S. P. Reiss. “PECAN: Program Development Systems that Support Multiple Views”. In: *IEEE Trans. Software Eng.* 11.3 (1985), pp. 276–285.

[RT84] Thomas Reps and Tim Teitelbaum. “The Synthesizer Generator”. In: *SIGSOFT Softw. Eng. Notes* 9.3 (Apr. 1984), pp. 42–48.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Boston, MA: Pearson, 2008.

[SF+09] David Svensson Fors, Boris Magnusson, Sven Gestegård Robertz, Görel Hedin, and Emma 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.

[TR81] Tim Teitelbaum and Thomas W. Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment". In: *Commun. ACM* 24.9 (1981), pp. 563–573.

[VSK89] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. "Higher-Order Attribute Grammars". In: *PLDI*. ACM, 1989, pp. 131–145.

[Völ09] Markus Völter. "MD* Best Practices". In: *Journal of Object Technology* 8.6 (2009), pp. 79–102.

Live Programming of Internet of Things in PalCom

Abstract

PalCom is a middleware toolkit for pervasive computing and internet-of-things. We discuss how PalCom supports exploration and live programming through three phases: exploring services, assembling them into applications, and exposing them as new services. We give an example of this workflow through the construction of a simple photo booth application.

1 Introduction

In pervasive computing, including Internet of Things (IoT), software applications are distributed, making use of many different services on different kinds of devices, and communicating over different underlying networks. Live programming can play a key role in programming such systems, allowing developers to explore the available devices and their services, and experiment with how to combine things and how to automate tasks.

PalCom [SF+09] is a middleware toolkit, designed to support *palpable computing*, a variant of pervasive computing where devices are made explicit (palpable). The toolkit is used in advanced home care applications [JM16], but is still under constant development.

In this paper, we identify key activities for live programming in PalCom, including *exploring* services, *assembling* them into partial applications, and *expos-*

Alfred Åkesson, Mattias Nordahl, Görel Hedin, and Boris Magnusson. "Live Programming of Internet of Things in PalCom". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Nice, France, 2018, pp. 121–126

ing new services from such assemblies. These partial applications can again be explored and assembled into larger applications.

We start with giving some background on PalCom (Section 2). Then we discuss live programming (Section 3) and give an example of how it is used in constructing a simple photo booth application (Section 4). We end with related work (Section 5), and conclusions (Section 6).

2 The PalCom Middleware Toolkit

PalCom is a service-based middleware toolkit. It provides an automatic discovery protocol which lets devices announce themselves and the services they provide, as well as to find other devices and their services. Through an abstraction of underlying network technologies, devices can communicate over different media, e.g., IP, Bluetooth, IR, or local in-memory communication between processes. This media abstraction makes it easy to build diverse, heterogeneous networks of devices, and a built-in routing protocol allows multiple such networks to be interconnected.

PalCom services communicate by asynchronously sending and receiving commands, but are agnostic of whom they communicate with. Each service defines its own commands, which serves as an API for communicating with it. To combine two or more services, an *assembly* is used, i.e., a script that connects to the services, and coordinates messages between them. Metaphorically, an assembly can be thought of as an adapting multiway cable that plugs into the services it combines. The assembly can itself provide new services, metaphorically corresponding to the adapting cable itself having a port that other assemblies/cables can plug into.

When an assembly is started, it automatically connects itself to the services it uses. So metaphorically, this is like the cable automatically locating the service ports and plugging itself in. The PalCom middleware takes care of automatically reconnecting in case parts of the network have been temporarily unavailable, e.g. when a mobile phone has been out of reach of its cellular network.

Figure 1 shows an example application for a photo booth. The preview assembly coordinates a button, a web camera, and a photo viewer service, and the print assembly coordinates a second button, a service on the preview assembly, and a print service. In Section 4, we will show how this application is constructed. The separation of functionality (in the services) and the coordination and configuration (in the assemblies), allows services to be reused for different purposes in different applications.

A running assembly knows exactly which set of services (and on which devices) it should connect to, so it can itself be run on any device in the network. This is why we don't show which devices the assemblies run on in Figure 1. They could run on, for example, a tablet on the same network.

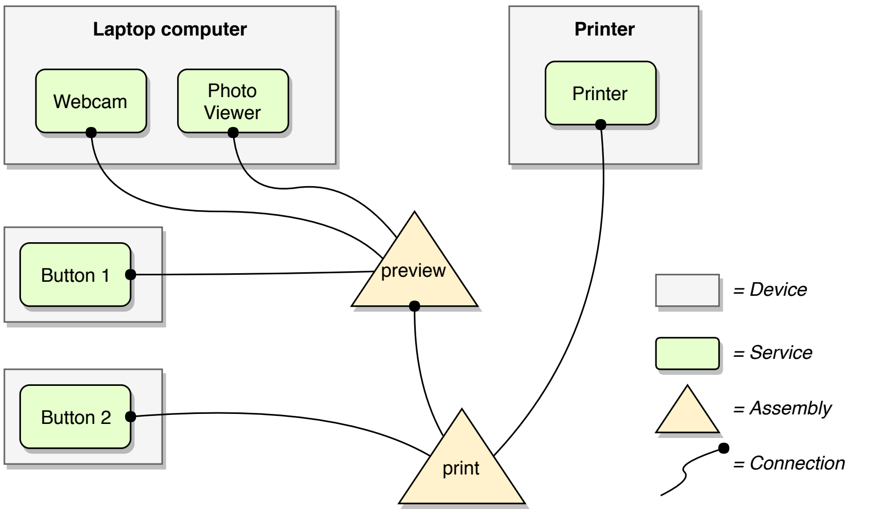


Figure 1: Overview of a PalCom photo booth application with two assemblies.

The assemblies are specified in a domain-specific language. An interactive tool called the PalCom *Browser* allows users to view discovered devices and services on the network, and also to create and edit assemblies using a projectional editor. The user can run the assembly directly in the browser tool, or export it in order to install it on another device.

3 Live programming in PalCom

To program an application, the developer can connect and script live services on live devices, using the PalCom browser.

In general, the developer starts by *exploring* how available services work. The browser shows the available devices and their services, and for each service what input- and output commands it has, i.e., its message protocol. To explore how a service works, the developer can bring up a remote interaction view, allowing direct interaction with the service, i.e., sending commands to it and viewing its response. While there may be documentation available for the commands, direct interaction with the service typically gives a much improved understanding of its dynamic behavior.

To combine services and automate tasks, the developer can write an assembly script. The assembly connects to other services and can send and receive messages from those services. Its script runs in an infinite loop, reacting to incoming messages in sequential order. The script is programmed mostly through drag-and-drop actions, dragging input- and output commands from the discovery view to the script view in order to specify the assembly's behavior. The script can also be edited using a projectional editor, e.g., to make use of conditionals and local variables. The developer can switch between running and editing the assembly, to check that it works in the intended way.

It is possible to *expose* functionality of an assembly, so that it can itself be connected to by other assemblies. This is done by defining a *synthesized* service of the assembly, with input and output commands. The assembly can receive input commands and send output commands through this service interface. When the assembly runs, its synthesized services appear in the discovery view like regular services, allowing them to be explored using remote interaction, as well as being used in new assemblies. This way, applications can be extended easily.

Figure 2 illustrates the activities of live programming in PalCom, showing that the user can go between these different activities in the development process.

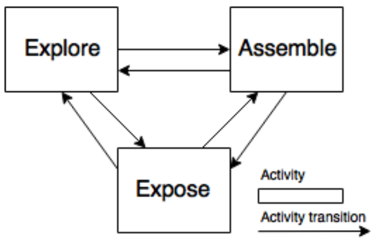


Figure 2: The activities of live programming in PalCom

4 Example: Photo booth

In this section, we will show how to use PalCom to program a photo booth application by combining off-the-shelf equipment like buttons and web cameras. We assume that all the equipment is running the PalCom middleware, and that all devices are connected to at least one of the interconnected networks.

The intended use of the photo booth is as a guestbook alternative at parties. The photo booth allows guests to go inside to take some photos, print one of them and hang the resulting photo on a wall. Here is a list of the equipment we use in this application:

- Laptop, with a photo viewer and a webcam service
- Printer
- Two separate Buttons communicating over Bluetooth.

Our system should work in the following way; a party guest sits in front of the laptop and presses one of the buttons. The camera will then take a photo and

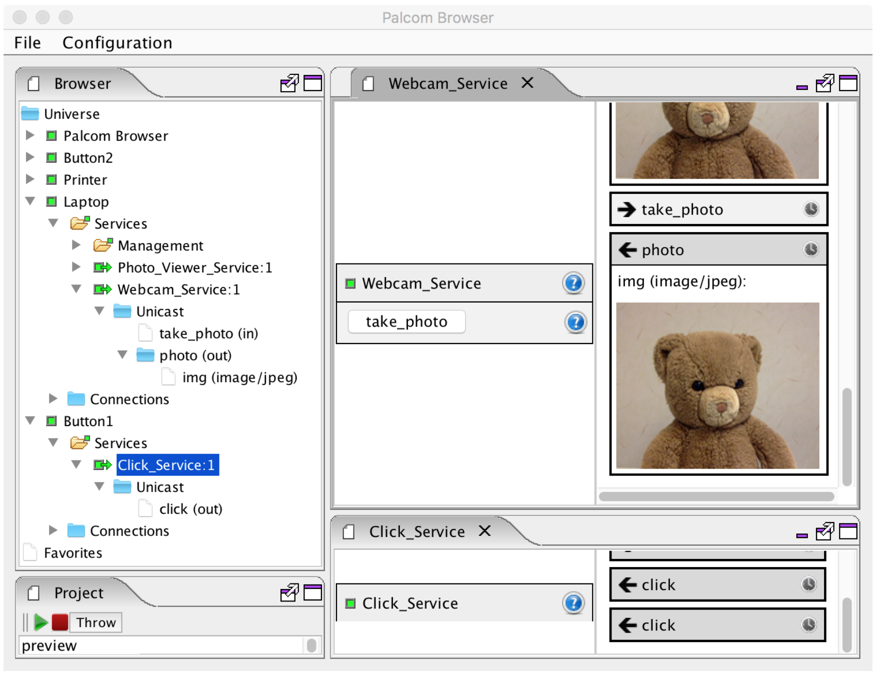
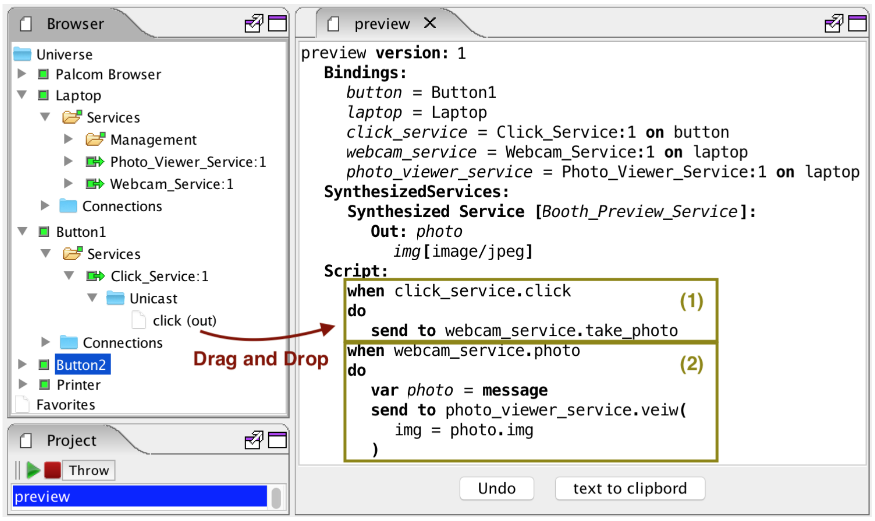


Figure 3: The PalCom Browser. To the left, the discovery view with devices, services, and commands. To the right, two remote interaction views. One for the web camera, and one for the click service on one of the buttons.



```
preview version: 1
  Bindings:
    button = Button1
    laptop = Laptop
    click_service = Click_Service:1 on button
    webcam_service = Webcam_Service:1 on laptop
    photo_viewer_service = Photo_Viewer_Service:1 on laptop
  SynthesizedServices:
    Synthesized Service [Booth_Preview_Service]:
      Out: photo
        img[image/jpeg]
  Script:
    when click_service.click
    do
      send to webcam_service.take_photo
    when webcam_service.photo
    do
      var photo = message
      send to photo_viewer_service.veiw(
        img = photo.img
      )
```

Figure 4: The PalCom Browser in assembly code editing mode

show it on the screen. The guest can take as many photos as he/she likes and preview them on the laptop screen. When the guest is satisfied with the photo, he/she presses the other button, and the last photo is printed.

One way of constructing this system is to connect the parts according to the overview shown in Figure 1, where the preview assembly handles the taking and previewing of the photo, and the print assembly controls the printing of the photo. Arriving at this solution involves a number of steps using live programming activities in the PalCom browser.

4.1 Explore

Our first goal is to try to connect one of the buttons to the camera. We begin by using the browser to explore how the webcam service works, see Figure 3. In the discovery view (to the left), we can find discovered devices and their services, as well as the commands of the services. Expanding the discovery information for the laptop device reveals the webcam service. Double-clicking on the webcam service opens a remote interaction view (to the right). Here, we can explore its capabilities. In this case, *Webcam_Service* has only one input command, *take_photo*. By clicking on this command, the command *take_photo* is sent to the camera service, to which its response is to take a photo and send it back. The right-hand part of the remote interaction view shows the history of commands sent to and received from the camera. Here we can see the actual photo, which is a parameter of the received photo command.

The next step is to explore the button. In the browser, we open the remote interaction view for the click service on *Button 1*. Here we can see that it has no input commands. On the other hand, we have a physical button that we can press. As seen in Figure 3, pressing the button results in its click service sending a *click* command.

4.2 Assemble

After exploring how the button and camera work, we can take the next step and combine them, using an assembly.

From the browser, we create a new assembly script, here called *preview*, see Figure 4. We would like the camera to take a photo whenever the button is pressed. To script this, we simply drag the *click* command from the discovery view into the editor view. An event handler `when ... do ...` is then automatically created, listening to the click command from the *Click_Service*. We then drag the *take_photo* command from the discovery view to the `do` part of the event handler to complete the desired behavior, namely that whenever the button sends a click command, the assembly will detect this, and send a take photo command to the camera. The resulting code is shown in block (1) in Figure 4.

When doing the drag-and-drop of commands into the script, local declarations of the involved device and service instances are automatically created under the `Bindings` heading in the script. The developer can edit the script to rename them to more suitable names if desired as done in the examples here.

We can now run the assembly to verify that it works as intended. Once started, the assembly connects to the camera and click services as declared in its bindings, and when the button is pressed `take_photo` commands are indeed sent to the camera, which replies with `photo` commands. In this case, the camera service is implemented in such a way that it sends the taken photo to all connected parties, i.e. both to the assembly (which for now, does nothing with it) and the Browser's remote interaction view.

4.3 Extending the Assembly

The assembly is not yet complete. We want to connect the photo viewer as well, so that each photo the camera takes is shown in the viewer. So we explore the capabilities of the photo viewer by opening a remote view for it. We can then observe that it has one input command that takes a photo as a parameter. We can try out this command by invoking it with a photo selected from our computer's file system as the parameter. There is no command sent back. Instead, the photo viewer service shows its latest received photo in a window on its hosting laptop.

After exploring the photo viewer, we extend the assembly script to add a new event handler that listens for the photo from the webcam and forwards the received photo to the photo viewer. Adding the event handler and sending the photo is done by two drag-and-drop edit actions. After dropping the viewer's photo command, a placeholder for its *img* parameter is generated, to which the image from the camera service (`photo.img`) can be assigned, by selecting it from a menu. The resulting code is shown in block (2) in Figure 4. The `when` clauses need to be mutually exclusive, and all incoming commands are handled in sequence. All the actions in a `do` section are executed in sequence.

We can now test run the preview assembly, and observe that every time we press the button, a new image is shown in the photo viewer's window.

4.4 Expose

We are now half-way finished building our photo booth. The remaining part is to be able to print the latest viewed photo. We would like to do this by pressing the second button.

One way of handling this is to let the preview assembly expose the photo just received from the camera, using a new service defined on the assembly, a so called *synthesized* service. We can then construct an additional assembly, *print*, that connects to this synthesized service and combines it with the second button and the printer.

Figure 5 shows how a synthesized service has been added to the script (*Booth_Preview_Service*), with an output command *photo*. The event handler that receives the photo from the web camera has also been extended with an additional action: to send the photo command out from the booth preview service, with the received photo as its parameter. If there are other assemblies that are connected to the booth preview, they will receive this command.

4.5 Creating the Print Assembly

Now, we can complete the photo booth application by creating the print assembly that combines the preview assembly with Button 2 and the printer.

We start by exploring the *Booth_Preview_Service* service that the preview assembly now exposes, again using a remote view. The service appears in the discovery view, like any other service, and we can observe how *photo* commands appear in the remote view every time we press the first button, see Figure 6.

We now create the print assembly, and start by adding an event handler that saves the latest preview photo in a transient variable *current_photo*. A transient variable is a global variable that can be assigned and accessed from all event handlers. If the assembly is restarted the transient variable is set to the default value given in its declaration, which in this example is empty.

Should the variable still be empty when used as a parameter, it will simply be up to the receiving service how it should be handled. Next, we add a second event handler that sends the currently saved photo to the printer whenever the second button is pressed. We can test run the assembly to make sure it works, and use the

```
preview version: 1
  Bindings:
    button = Button1
    laptop = Laptop
    click_service = Click_Service:1 on button
    webcam_service = Webcam_Service:1 on laptop
    photo_viewer_service = Photo_Viewer_Service:1 on laptop
  SynthesizedServices:
    Synthesized Service [Booth_Preview_Service]:
      Out: photo
        img[image/jpeg]
  Script:
    when click_service.click
    do
      send to webcam_service.take_photo
    when webcam_service.photo
    do
      var photo = message
      send to photo_viewer_service.veiw(
        img = photo.img
      )
      send from Booth_Preview_Service.photo(
        img = photo.img
      )
```

Figure 5: The preview assembly after adding the synthesized service

remote views to explore its different parts. The resulting photo booth assembly is shown in Figure 7.

4.6 Possible Extensions

We have shown how services can be explored live, and their functionality combined and coordinated by assembling them into useful applications. The photo booth application could be extended further by incorporating other services, e.g., allowing guests to sign their photos, adding different image filters to them, or, in addition to printing the photos, connecting the assembly to a service in the cloud to publish the photos on a website. Regardless of what other services are available on the network, they too can be explored and assembled as just shown.

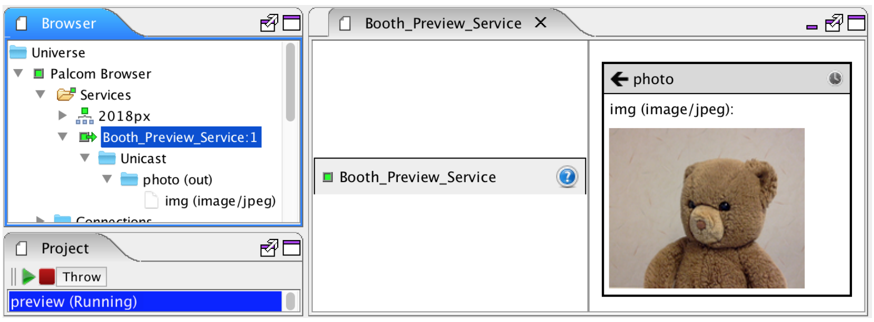


Figure 6: The remote interaction view for the synthesized service

5 Related Work

PalCom programming is inspired by the idea of programming by example [Hal84], in that the interactions programmed in an assembly relate to existing physical or virtual example objects. The idea of programming-by-example might be pushed even further by adding support for using recorded interactions in the remote views to generate parts of the assembly scripts.

For physical objects like buttons and cameras, as in our photo booth example, an interesting avenue of further research might be to make use of actual physical interaction in order to build assemblies. In our photo booth example, the user might, for example, actually click on the button instead of doing a drag-and-drop in the browser, or interacting with it via the remote view. For physical objects that do not have built-in interaction like buttons, other interaction techniques might be investigated, for example, the PICOntrol handheld projector suggested by Schmidt et al. [SMC12]. However, in most PalCom scenarios, there are also many services that are virtual rather than physical. Examples are the assemblies themselves, as well as purely computational services, and web services in the cloud. Furthermore, physical devices do not necessarily need to be locally available. They might be in the next room, or in a completely different place. Making use of actual physical interaction would therefore need to integrate in a smooth way also with remote and virtual services.

Programming in PalCom can be compared to the read-eval-print loop (REPL) used in many programming environments. Findler et al. discuss using the REPL in the context of DrRacket and Scheme [Fin+97], allowing the programmer to interact with and explore a program. After the developer has tried out some things in the REPL, he/she can change the original program and begin to experiment with the new version of the program. This is similar to how the developer can explore services in a PalCom network, and then change an assembly program to create a new version of it. A difference from general-purpose programming is that a central part of PalCom programming is to interact with live devices and services.

```
print version: 1
  Bindings:
    browser = Palcom Browser
    button2 = Button2
    printer = Printer
    booth_preview_service = Booth_Preview_Service:1 on browser
    click_service = Click_Service:1 on button2
    printer_service = Printer_Service:1 on printer
    transient current_img [image/jpeg] =
  SynthesizedServices:
  Script:
    when booth_preview_service.photo
    do
      var photo = message
      current_img = photo.img
    when click_service.click
    do
      send to printer_service.print(
        data = current_img
      )
```

Figure 7: The print assembly, which connects the preview service from the preview assembly with the printer and the second button.

Another relevant comparison is to the idea of liveness, as described by Tanimoto [Tan90][Tan13], referring to the ability to modify a running program. He introduced four levels of liveness in 1990, going from an ancillary description, to being *fully live*. Later he expanded the model with two more levels which additionally includes prediction[Tan13]. The levels primarily apply to general-purpose programming, but a tangent can be drawn to the live programming of assemblies in the PalCom Browser. At liveness level four, a program can be modified whilst running and will immediately reflect the change in its behavior and output. Similarly, an assembly being updated will immediately change the behavior and output, but for an entire distributed system (or parts of it), rather than a single program. Going back to the photo booth example, if the event handler for button 2 was changed to publish photos to a cloud-based gallery rather than printing them, this new behavior would immediately apply the next time the user pressed the button. It is worth noting, though, that since PalCom's communication is distributed and event based, while the behavior of the system is immediately changed, the effect of the change, and thus user feedback, only becomes apparent once a new event occurs, e.g. by pressing the button. In this respect, assembly editing might not fulfill the requirements for Tanimoto's fourth liveness level.

On the surface, the interaction using Drag and Drop for the Palcom Browser look similar to the interactions used in Scratch[Mal+10]. Scratch is a visual block programming language designed for children. In Scratch, you drag from a list of blocks concerning the current program whereas in the Palcom Browser you drag from a dynamic list of devices and services currently present on the network. When you drop a block in Scratch, it creates a new instance of that block, but when you drop a command in the Palcom Browser, it creates a reference to that command.

OSCAR [NES08] is another IoT system supporting service composition. It focuses specifically on supporting end users, employing an intuitive user interface. The system allows users to connect media streams, for example, connecting a video stream from a web camera to a particular TV screen. There is a composition concept called a *Setup* which is an end-user programmed connection between two devices, and where the endpoints (the actual devices) can be dynamically selected based on rules. PalCom assemblies have a different focus, namely to coordinate a number of services, and supporting event-based communication. Furthermore, assemblies can contain logic in order to program multi-step transactions, and they can expose new services to be used as building blocks for other assemblies.

6 Conclusions

We have discussed how live programming of IoT applications is done in PalCom, by exploring live services and gradually assembling them into applications where parts and assembled parts can be test run and changed in an exploratory fashion. Exposing partial applications as new services allows the same kind of exploration, using remote views, to be done as for general purpose programmed services.

To illustrate this way of live programming, it was discussed in detail how to construct a simple photo booth example.

We are currently experimenting with the syntax of the assembly language, and with naming conventions, in order to get more intuitive scripts. We are also looking into running user experiments to guide our language design. We are also experimenting with how to package applications as configurations of versioned assemblies and services, in order to deploy and update them easily. Furthermore, we will look into different message exchange patterns and ways to visualize and analyze a running system of several connected services and devices.

7 Acknowledgements

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We thank Christoph Reichenbach and the anonymous reviewers for helpful comments on earlier drafts of the paper. We also thank the PX workshop participants for valuable feedback.

References

[Fin+97] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. “DrScheme: A pedagogic programming environment for scheme”. In: *Programming Languages: Implementations, Logics, and Programs*. Ed. by Hugh Glaser, Pieter Hartel, and Herbert Kuchen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 369–388.

[Hal84] Daniel Conrad Halbert. “Programming by example”. PhD thesis. University of California, Berkeley, 1984.

[JM16] Björn A Johnsson and Boris Magnusson. “Supporting collaborative healthcare using PalCom–The itACiH system”. In: *Pervasive Computing and Communication Workshops (PerCom Workshops), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–6.

[Mal+10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. “The Scratch Programming Language and Environment”. In: *Trans. Comput. Educ.* 10.4 (Nov. 2010), 16:1–16:15.

[NES08] Mark W. Newman, Ame Elliott, and Trevor F. Smith. “Providing an Integrated User Experience of Networked Media, Devices, and Services through End-User Composition”. In: *Pervasive Computing*. Ed. by Jadwiga Indulska, Donald J. Patterson, Tom Rodden, and Max Ott. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 213–227.

[SMC12] Dominik Schmidt, David Molyneaux, and Xiang Cao. “PICOntrol: Using a Handheld Projector for Direct Control of Physical Devices Through Visible Light”. In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST ’12. ACM, 2012, pp. 379–388.

[SF+09] David Svensson Fors, Boris Magnusson, Sven Gestegård Robertz, Görel Hedin, and Emma 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.

[Tan90] Steven L. Tanimoto. “VIVA: A visual language for image processing”. In: *Journal of Visual Languages & Computing* 1.2 (1990), pp. 127 –139.

[Tan13] Steven L Tanimoto. “A perspective on the evolution of live programming”. In: *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press. 2013, pp. 31–34.

COMPOS: Composing Oblivious Services

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.

1 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 [DEDP15; Che+14; PLM14]. Also, future IoT applications are expected to contain more powerful devices, with more computation taking place at the edge of

Extended version of Alfred Åkessson, Görel Hedin, Boris Magnusson, and Mattias Nordahl. "ComPOS: Composing Oblivious Services". In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. Kyoto, Japan, Mar. 2019, pp. 132–138

the network, and need to handle unreliable (weak) connectivity of heterogeneous networks in a robust way [TM17].

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 [SF09; SF+09; ÅNHM18b] 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 [Smi87]. 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.

This report 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 [SF09], 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 *reactions* that add state to the compositions. In this report, we choose to remove old

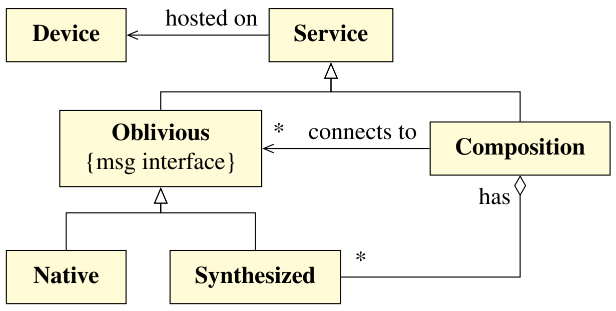


Figure 1: Conceptual model for PalCom devices and services.

reactions when new arrive, and moreover we will motivate this choice. A main design goal is to make the language 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.

In the following sections, we first present a motivating example for composing services (Section 2). We then introduce our DSL, explaining the features and the interpreter (Section 3). To exemplify more features of the DSL we do a couple of extensions to the example (Section 4). Further, we change how a composition handles spontaneously incoming messages. We do this by adding an extra service and changing the composition (Section 5). 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 6). Finally, we discuss related work (Section 7), and end with conclusions and prospectives for future research (Section 8).

2 Motivating example

As a motivating example, we will use a variant of a bird watching scenario [ÅNHM18a], and discuss how to construct a supporting application using COMPOS.

2.1 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.

2.2 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
- 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.

2.3 Composing the application

To construct the bird watcher application from the above services, Maria creates a *composition* (a COMPOS script), that connects to the relevant services, and that includes a script for how messages should be mediated. Figure 3 shows an overview of the system and Figure 4 shows the composition script and a corresponding sequence diagram. When a *move* message arrives from the motion

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

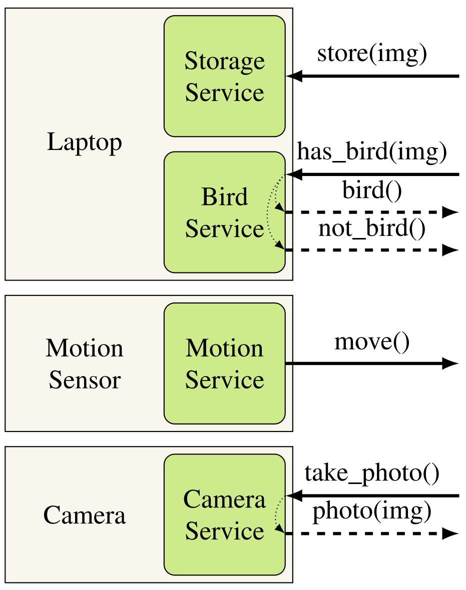


Figure 2: Services (green boxes with rounded corners) running on devices (white boxes with sharp corners). Commands and requests (solid arrows), responses (dashed arrows).

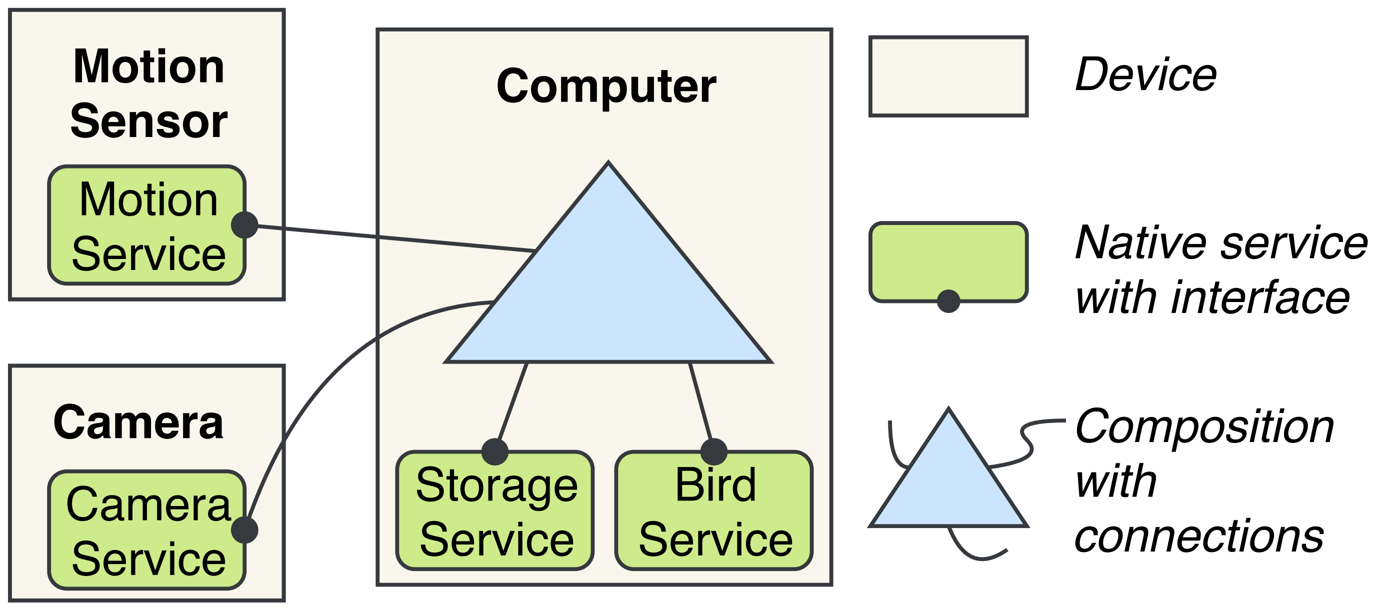
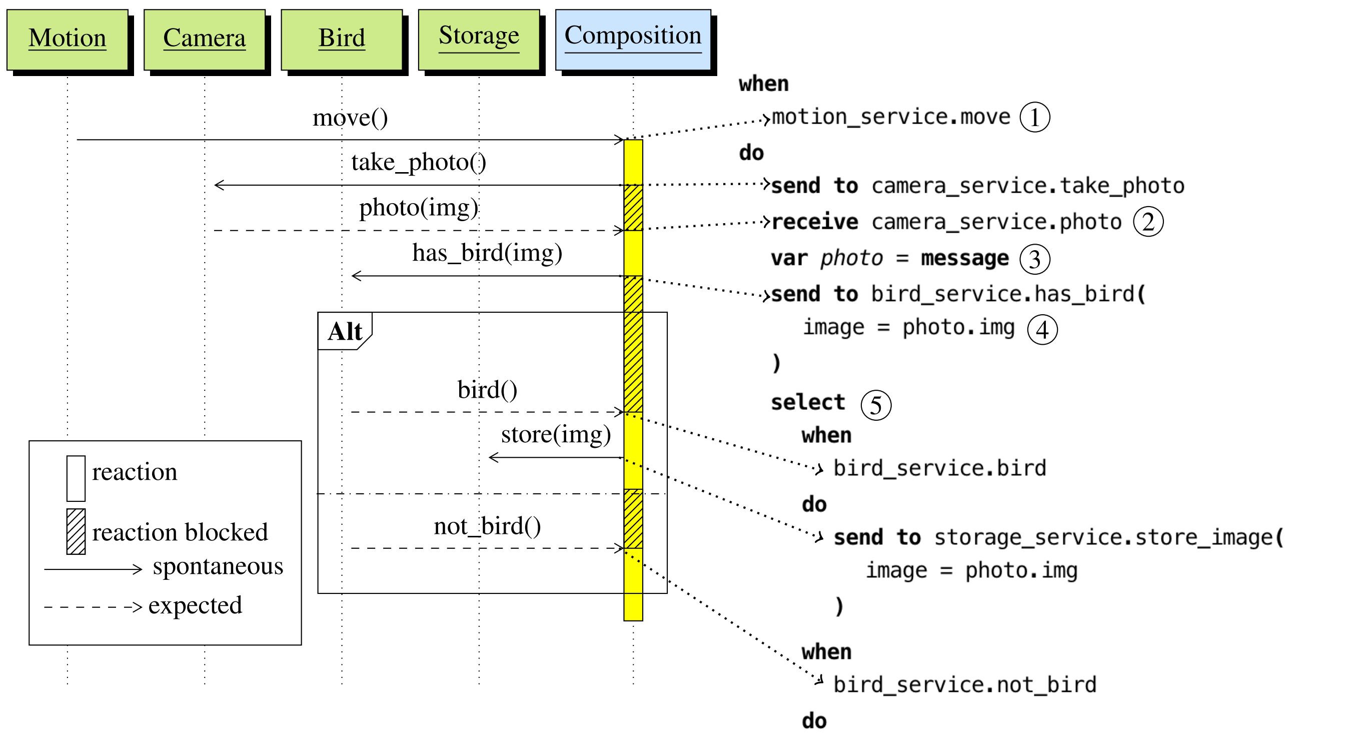


Figure 3: Overview of the Bird watcher system showing the connections between services and the composition.



```
when
  motion_service.move ①
do
  send to camera_service.take_photo
  receive camera_service.photo ②
  var photo = message ③
  send to bird_service.has_bird(
    image = photo.img ④
  )
  select ⑤
    when
      bird_service.bird
    do
      send to storage_service.store_image(
        image = photo.img
      )

    when
      bird_service.not_bird
    do
```

Figure 4: 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. See the full script in appendix B.1.

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 4, see appendix B.1). 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 [ÅNHM18b; ÅH18].

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.

3 The ComPOS language

3.1 Coordination constructs

A ComPOS coordination script consists of a set of guarded actions, so called *when-do*s. The *when* part contains a service reference and a message name, see (1) in Figure 4 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-specification*, 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 4. A more in depth description of the syntax and semantics of the language can be found in appendix A.

3.2 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 a queue for incoming messages, and handles the messages in order of arrival. When receiving a message that matches a when-do clause at the outermost level, the interpreter creates a new *reaction* for the corresponding reaction-specification. The reaction 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 current reaction until the reaction blocks or finishes. If the reaction blocks, it 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 reaction, the interpreter continues to execute that reaction. A received message can match at most one outer when-do or blocked reaction, and gets ignored if it has no match. Messages sent to connections that are currently down, are by default lost.[2] Pseudocode for the interpreter can be found in appendix A.3.

3.3 Spontaneous messages during reactions

A blocked reaction 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*? Possible ways of dealing with this situation are:

parallel Start a parallel reaction for the new message.

queue Queue up the message and start its reaction when the ongoing reaction has completed.

ignore Ignore the new message.

abort Abort the current reaction and start a new one.

In our interpreter, we have chosen the *abort* option: abort the current reaction, and start a new reaction for the new message. This way we avoid old reactions that remain indefinitely, which would happen for the options *parallel*, *queue*, and *ignore*,

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

and we avoid having an unbounded number of simultaneous reactions that might arise from parallelisation. Furthermore, *abort* will prioritize the latest information, in contrast to queueing and ignoring. In some situations, one of the options *parallel*, *queue* or *ignore* are more suitable. By choosing to abort we can build the other options on top. To create the behaviour of these three options, we relay incoming messages to a message strategy service (see section 5).

3.4 Separating reactions related to remote reactions

Sometimes, incoming spontaneous messages on the same connection are independent, and these messages should not abort each other's reactions. This is the case when two messages of the connection originate from different services. To handle this, each reaction is associated with the connection and the *reaction id* of its first message, called the *remote-reaction id*. Messages on the same connection but with different remote-reaction ids result in independent reactions that do not abort each other. This way, many ongoing reactions over the same connection can be handled. Examples where this happens are given in sections 4.3 and 5.3. Note that messages matching different guards abort each other if they come on the same connection, and they have the same remote-reaction id.

4 Extending the bird-watcher scenario

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

4.1 Composing compositions using synthesized services

A synthesized service [SF+09] is an abstraction mechanism that allows a composition to provide functionality in the form of oblivious services. 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, Combine␣Bird␣Services, 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 5 (right) or appendix B.2.2. 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.

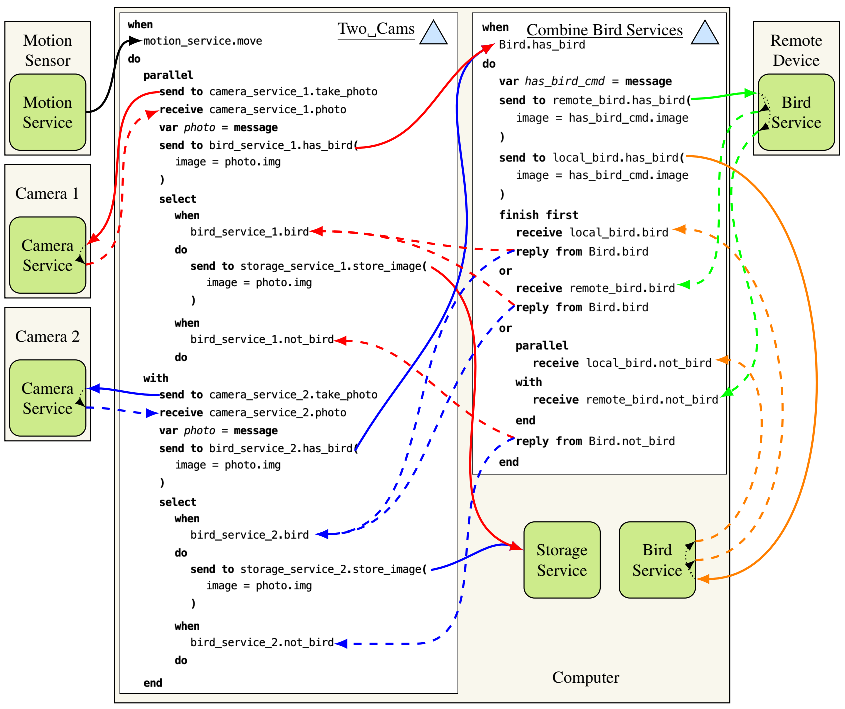
4.2 Reactions triggered from multiple connections

Suppose now that Maria wants to add one more camera to her system, to see the birds from more angles. She modifies her bird watcher composition and uses the parallel action to allow both cameras to take and process pictures in parallel. She calls this modified version of the composition Two␣Cams, see Figure 5 (left) or appendix B.2.1. This composition uses the synthesized service Bird to use the combined local and remote bird-recognizing services. See Figure 6 for an overview of the system.

In Two␣Cams, she sets up two connections to the same Combine␣Bird␣Services composition, allowing it to differentiate between the different requests. This way, the Combine␣Bird␣Services composition will create one reaction for each request instead of letting them abort each other. It would be desirable to be able to avoid multiple connections to the same service. In future work, we will investigate how this could work, in particular for identical requests from parallel branches.

4.3 Duplicated code factored out to new composition

The Two␣Cams composition has duplicated code that Maria would like to avoid. She dose this by refactoring out the duplicated code into a new composition called Store␣Bird␣Image with a synthesized service Store␣If␣Bird. Two␣Cams now sends the pictures to Store␣If␣Bird, using two different connections. Because of the two different connections, pictures from the two cameras will start different reactions at Store␣Bird␣Image, without aborting the other. Each of these reactions has a unique reaction id, and messages they send will therefore not abort each other, even if they are sent on the same connection. Figure 7 shows an overview of the system, pointing out the connections and reaction ids. See appendix B.3 for a listing of the Store␣Bird␣Image composition and the refactored version of Two␣Cams using Store␣If␣Bird.



```
when
  motion_service.move
do
  parallel
    send to camera_service_1.take_photo
    receive camera_service_1.photo
    var photo = message
    send to bird_service_1.has_bird(
      image = photo.img
    )
    select
      when
        bird_service_1.bird
      do
        send to storage_service_1.store_image(
          image = photo.img
        )
      when
        bird_service_1.not_bird
      do
  with
    send to camera_service_2.take_photo
    receive camera_service_2.photo
    var photo = message
    send to bird_service_2.has_bird(
      image = photo.img
    )
    select
      when
        bird_service_2.bird
      do
        send to storage_service_2.store_image(
          image = photo.img
        )
      when
        bird_service_2.not_bird
      do
  end
```

```
when
  Bird.has_bird
do
  var has_bird_cmd = message
  send to remote_bird.has_bird(
    image = has_bird_cmd.image
  )
  send to local_bird.has_bird(
    image = has_bird_cmd.image
  )
  finish first
    receive local_bird.bird
    reply from Bird.bird
  or
    receive remote_bird.bird
    reply from Bird.bird
  or
    parallel
      receive local_bird.not_bird
    with
      receive remote_bird.not_bird
    end
    reply from Bird.not_bird
  end
```

Figure 5: Compositions used in the extended scenario. Two␣Cams (left) is a modified version of the original bird watcher composition. Combine␣Bird␣Services (right) is a composition of two different bird recognizers. An initial *move* message (black) to Two␣Cams leads to two parallel subreactions (red and blue), leading to two corresponding reactions in Combine␣Bird␣Services.

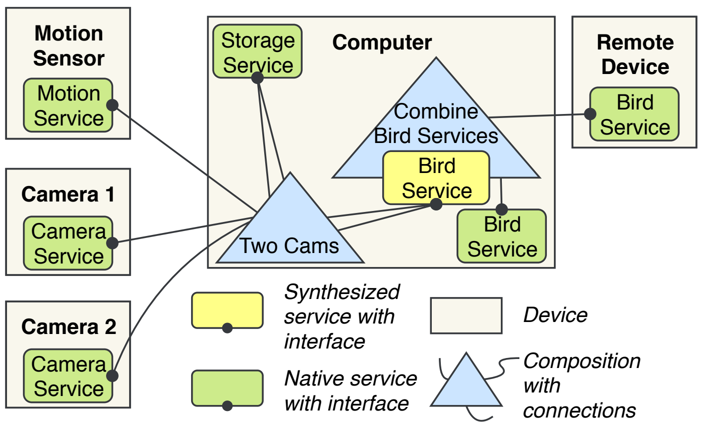


Figure 6: Overview of the two camera system before refactoring.

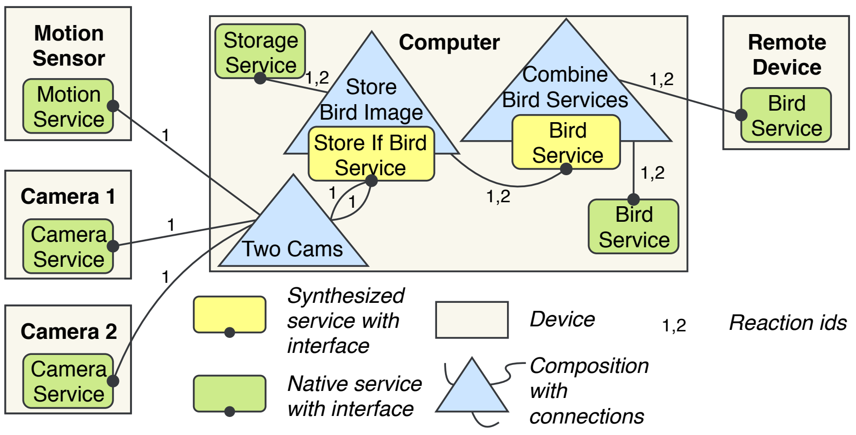


Figure 7: An overview of the system after refactoring, illustrating the connections and reaction ids.

5 Adapting semantics for spontaneous messages

We chose the *abort* strategy from 3.3 as default, but sometimes another strategy is preferable. In this section we describe how we can implement *ignore*, *queue* and *parallel* from section 3.3. This is done by adding a message strategy service and changing the composition. Figure 8 shows an overview of the original one-camera system with an added strategy service.

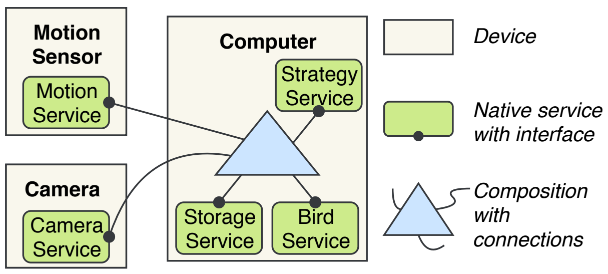


Figure 8: An overview of the system where a strategy service is used, for example, a latch

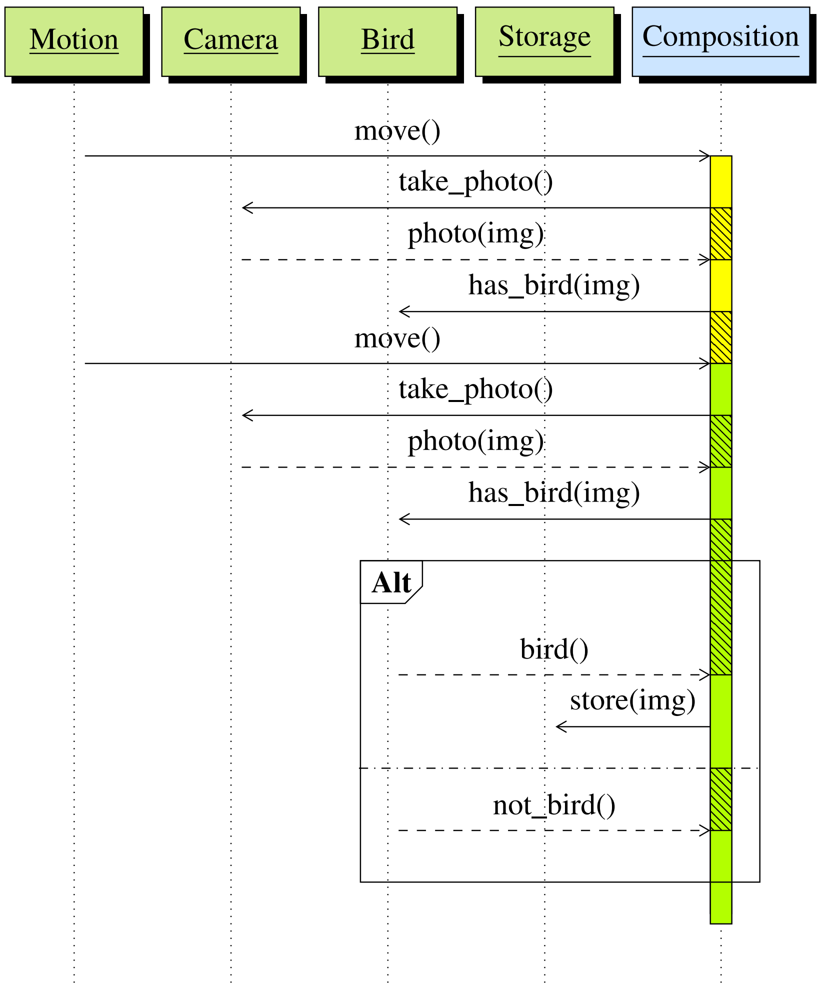


Figure 9: The sequence diagram shows the move command aborting the currently blocked reaction (yellow) and initiating a new one (green).

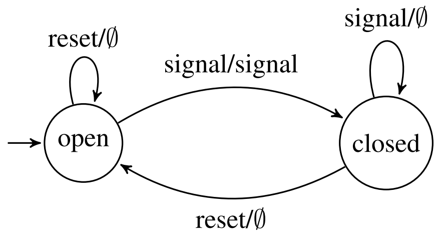


Figure 10: State machine for the latch service. The labels of the edges have the form (received message/sending message) where ∅ means sending no message.

5.1 Ignoring spontaneous messages

Sometimes we want to be able to ignore the spontaneous messages that arrive during an reaction. For example, in Figure 9 we see another sequence diagram from Maria's system. In this diagram, we see how the second *move* message aborts the old reaction and creates a new one. The interpreter has undesired default behaviour in this situation. If the motion sensor sends move messages with a high frequency, no reaction will be able to finish. This is because a new move message will abort the ongoing reaction before it has finished.

To deal with this undesirable behaviour, Maria wants a reaction always to finish. She wants the composition to ignore move messages arriving during a reaction. This means having the composition work like *ignore* in 3.3. To do this, she uses a *latch* service. The latch service implements the state machine shown in Figure 10. The first time the latch service receives a *signal* it sends a new signal as a spontaneous command. After that, the service ignores all incoming signals until it receives a *reset* command. After the reset, the latch service goes to its initial state. To use the latch service, Maria adds a *when-do* that every time it receives a move, it sends a signal command. She changes the other *when-do* to listen for a signal message and to send a reset message at the end of the reaction. See the full code in appendix B.4. The change of the composition allows a reaction to run to completion, without being aborted by any move message. Figure 11 shows a scenario where the latch pattern prevents the abortion of a reaction.

5.2 Queuing messages

Instead of ignoring incoming spontaneous messages, it may be desirable to queue them up, and treat them one at the time. We can implement this by using a similar approach to *ignore*, but using a queuing service instead of a latch service. The queuing service is like the latch service but instead of ignoring messages it puts them in a queue. When it receives a reset it sends the next message in the queue.

5.3 Parallelizing messages

When the motion sensor sends messages with a high frequency, we could instead of ignoring them handle the messages in parallel. In the *parallel* form, we would like each spontaneous command to start a new parallel reaction. To implement this semantics we use a *parallelizer* service. The parallelizer service has one *in* command and one *out* command. When the parallelizer service gets an *in* command, it sends an *out* command with a new reaction id. To use the parallelizer service we need to add a *when-do* like the one used in the latch pattern. This *when-do* listens for the message that we want to parallelize and sends an *in* command to the parallelizer service. We use the *out* command as the trigger for the reaction we want to parallelize. Figure 12 shows the use of a parallelizer service in a sequence diagram.

5.4 Discussion

In deciding the semantics for incoming spontaneous messages, we chose the abort solution from 3.3 as the default. This allows us to create all the other solutions using an extra service. We run the composition and the extra service on the same device to ensure that no messages between them get lost. We are considering adding syntactic sugar to allow the end user to express these patterns concisely. Today we have to create a custom variant of the extra service to enable it to forward messages of a specific type. In the future, we may add support for some kind of generics to parametrize the extra services with different message types.

6 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 5. 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[3].

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 1 shows the utility analysis of the two-camera system in Figure 5.

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

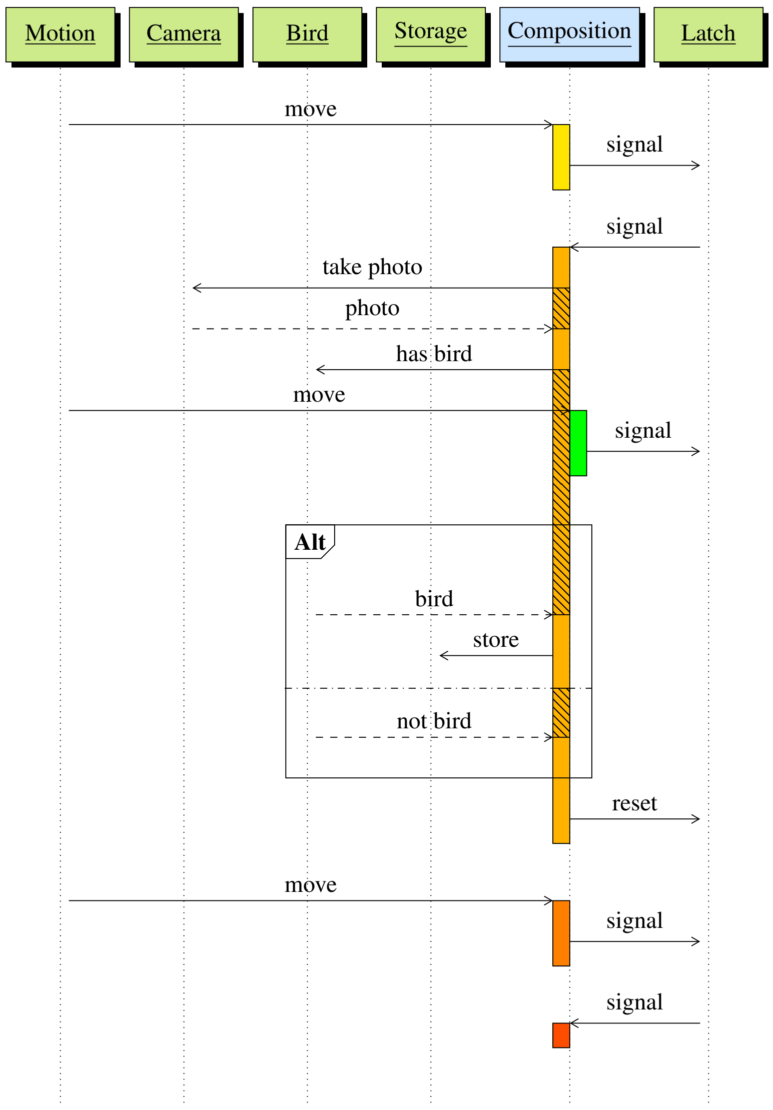


Figure 11: The sequence diagram shows the use of a latch service. When the first signal command arrives at the latch, the service sends a signal back to the composition. However, when the second signal arrives, the latch ignores it, and it does not send any command. When the third signal arrives at the latch, the latch has recently received a reset command, and a signal command is again sent to the composition. The use of the latch ensures that the reaction dealing with the photo always finishes.

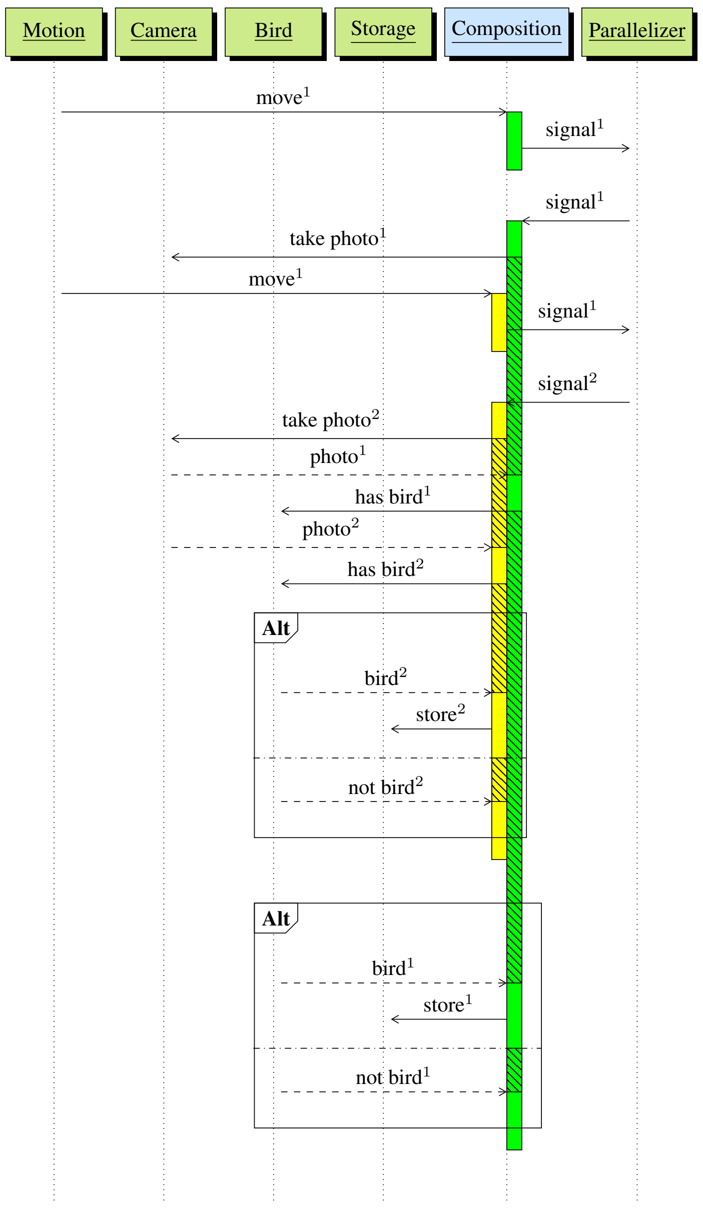


Figure 12: The sequence diagram shows the use of a parallelizer service. The superscript after the message name is the reaction id of the message. In the diagram, the parallelizer sends back a signal command with unique a reaction id each time. The composition interpreter starts a reaction for every signal command with a unique reaction id.

Table 1: A utility analysis of the system shown in Figure 5.

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 Two␣Cams composition creates a new reaction 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 Cameras	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	

From Table 1 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. If Maria also uses a latch service in her system, there would no longer be support for weak connectivity–the system would become useless as soon as one of the devices was no longer reachable. Having a timer that resets the latch would restore these properties, again making it useful even when it is only partially connected.

7 Related work

7.1 Previous composition languages for PalCom

Svensson, Hedin, and Magnusson [SHM07] 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 (4.0.19)[4]. Linus Åkesson [Åke16] 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 supports nested and parallel action sequences, but differs in the semantics of new spontaneous messages that arrive during a reaction. In Linus Åkesson's approach, new messages start a parallel reaction (option 1 in section 3.3), 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 Linus Åkesson's language is purely text-based, without any integration with a GUI, and is not intended for end users.

7.2 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 [MGZ14] and BPEL [Bar+07]. 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.

7.3 AmbientTalk

AmbientTalk [Cut+07] is a domain-specific language developed for programming message-based applications in mobile ad-hoc networks. It is thus similar to

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

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

7.4 End-user development for IoT

There are different approaches for end-user development of IoT systems. Some use programming by demonstration [LLCM17], whereas others use different types of DSL:s, like TeC [Sou+11], Midgar [GGBECF14], and AppsGate [CC16].

TeC [Sou+11] 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 [GGBECF14] 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 [CC16] 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.

8 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 2). 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

4). We show how to change the semantics for spontaneous messages with an extra service (Section 5). 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 6). 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 formalize the utility analysis and build a tool for it. To improve COMPOS, we want to generalise our language to support e.g. automatically parallelize requests to avoid multiple connections to the same service (see 4.2) and parametrization of services (see 5.4). We would also like to continue looking into the end-user programming perspective of COMPOS, and do user studies to evaluate its usability.

Acknowledgements

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

[ÅHMN19] Alfred Åkessson, Görel Hedin, Boris Magnusson, and Mattias Nordahl. “ComPOS: Composing Oblivious Services”. In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops).* Kyoto, Japan, Mar. 2019, pp. 132–138.

[Bar+07] Charlton Barreto, Vaughn Bullard, Thomas Erl, John Evdemon, Diane Jordan, Khanderao Kand, Dieter König, Simon Moser, Ralph Stout, Ron Ten-Hove, Ivana Trickovic, and Danny van der Rijn. *Web Services Business Process Execution Language Version 2.0.* Standard. OASIS, 2007.

[Che+14] Shanzhi Chen, Hui Xu, Dake Liu, Bo Hu, and Hucheng Wang. “A vision of IoT: Applications, challenges, and opportunities with China perspective”. In: *IEEE Internet of Things journal* 1.4 (2014), pp. 349–359.

[CC16] J. Coutaz and J. L. Crowley. “A First-Person Experience with End-User Development for Smart Homes”. In: *IEEE Pervasive Computing* 15.2 (2016), pp. 26–39.

[Cut+07] 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)*. 2007, pp. 3–12.

[DEDP15] Hasan Derhamy, Jens Eliasson, Jerker Delsing, and Peter 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.

[GGBECF14] Cristian González García, B Cristina Pelayo G-Bustelo, Jordán Pascual Espada, and Guillermo Cueva-Fernandez. “Midgar: Generation of heterogeneous objects interconnecting applications. A Domain Specific Language proposal for Internet of Things scenarios”. In: *Computer Networks* 64 (2014), pp. 143–158.

[LLCM17] Toby Jia-Jun Li, Yuanchun Li, Fanglin Chen, and Brad A. Myers. “Programming IoT Devices by Demonstration Using Mobile Apps”. In: *End-User Development*. Ed. by Simone Barbosa, Panos Markopoulos, Fabio Paternò, Simone Stumpf, and Stefano Valtolina. Cham: Springer International Publishing, 2017, pp. 3–17.

[MGZ14] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. “Service-Oriented Programming with Jolie”. In: *Web Services Foundations*. Ed. by Athman Bouguettaya et al. New York, NY: Springer New York, 2014, pp. 81–107.

[PLM14] Riccardo Petrolo, Valeria Loscri, and Nathalie 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.

[Smi87] Randall 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*. CHI ’87. ACM, 1987, pp. 61–67.

[Sou+11] João P Sousa, Daniel Keathley, Mong Le, Luan Pham, Daniel Ryan, Sneha Rohira, Samuel Tryon, and Sheri Williamson. “TeC: end-user development of software systems for smart spaces”. In: *International Journal of Space-Based and Situated Computing* 1.4 (2011), pp. 257–269.

[SHM07] D. Svensson, G. Hedin, and B. Magnusson. “Pervasive applications through scripted assemblies of services”. In: *IEEE International Conference on Pervasive Services*. 2007, pp. 301–307.

[SF09] David Svensson Fors. “Assemblies of pervasive services”. PhD thesis. Department of Computer Science, Lund University, 2009.

[SF+09] David Svensson Fors, Boris Magnusson, Sven Gestegård Robertz, Görel Hedin, and Emma 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.

[TM17] Antero Taivalsaari and Tommi Mikkonen. “A roadmap to the programmable world: software challenges in the IoT era”. In: *IEEE Software* 1 (2017), pp. 72–80.

[ÅNHM18a] Alfred Åkesson, Mattias Nordahl, Görel Hedin, and Boris Magnusson. “Demo: A DSL for composing IoT systems”. In: *Proceedings of the 19th ACM/IFIP Middleware Conference: Posters and Demos*. Rennes, France, 2018, pp. 17–18.

[ÅH18] Alfred Åkesson and Görel Hedin. “Jatte: A Tunable Tree Editor for Integrated DSLs”. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Comprehension of Complex Systems*. CoCoS 2017. Vancouver, BC, Canada, 2018, pp. 7–12.

[ÅNHM18b] Alfred Åkesson, Mattias Nordahl, Görel Hedin, and Boris Magnusson. “Live Programming of Internet of Things in PalCom”. In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Nice, France, 2018, pp. 121–126.

[Åke16] Linus Åkesson. *On the design of connector languages for latency-critical distributed applications.* Licentiate thesis 2016:1. Department of Computer Science, Lund University, 2016.

Appendix A A COMPOS Specification

A.1 Syntax

In this section we describe the projected syntax for COMPOS using W3C style EBNF[5]. Because COMPOS uses a projectional editor, the grammar does not need to be parsable and therefore contains some ambiguities. The grammar symbols are described in appendix A.2.

```
Composition ::=
    version
    "bindings:" bindings
    "synthesized␣services:" synthesized-service*
    "script:" script;
bindings ::=  device-variable*
    service-variable* transient-variable*;
device-variable ::= name-def "=" device-ref;
service-variable ::= name-def "=" service-ref
    "on" device-access;
transient-variable ::= "transient" name-def type
    ("=" <init-value>)?;
name-def ::= text-literal;
name-use ::= text-literal;
access ::=
      simple-access
    | qualified-access;
simple-access ::= name-use;
qualified-access ::= name-use "." access;
device-access ::= access;
service-access ::= access;
message-access ::= access;
type ::= "[" type-string "]";
synthesized-service ::= "synthesized␣service"
    name-def message-def*
message-def ::=
      in-message-def
    | out-message-def;
in-message-def ::= "in:" name-def parameter*
out-message-def ::= "out:" name-def parameter*
parameter ::= name-def type
script ::=  when-do*
when-do ::= "when" condition "do" branch;
condition ::=
      message-condition
    | service-connected
    | service-disconnected;
message-condition ::= message-access;
service-connected ::= service-access;
service-disconnected ::= service-access;
```

[5]https://www.w3.org/TR/xml/#sec-notation

```
branch ::= action*
action ::=
      local-variable
    | assign
    | receive
    | select
    | parallel
    | finish-first
    | send-to-service
    | send-from-synthesized;
local-variable ::= "var" name-def ("=" expr)?;
assign ::= access "=" expr;
receive ::= "receive" message-access;
select ::= "select" when-do*;
parallel ::=
    "parallel"
    (branch "with")*
    branch
    "end";
finish-first ::=
    "finish␣first"
    (branch "or")*
    branch
    "end";
send-to-service ::= "send␣to" message-access ("(" arg+ ")")?;
send-from-synthesized ::= send-type "from" message-access
    ("(" arg+ ")")?;
arg ::= access "=" expr;
send-type ::=
      send
    | reply;
send ::= "send";
reply ::= "reply";
expr ::=
      text-literal
    | latest-message
    | access;
latest-message ::= "message";
```

A.2 Grammar symbols

In this section, we describe most of the grammar symbols used in appendix A.1. Some of the grammar symbols are literals with an internal structure that is not editable with parts hidden from the user. For example, universally unique identifiers (UUIDs) of device and service references are hidden.

Symbol	**Description**
Composition	The root of the composition.
version	Contains the version information of the composition; it contains both UUID and readable name. The version is projected as the readable name of the composition and the readable name of the version.
bindings	Lists all global variables, including devices and services.
device-variable	Declares a local name for the device used in the composition.
device-ref	A reference to a device, including both a readable name and a UUID.
service-variable	Declares a local name for a specific service running on a specific device.
service-ref	A reference to a service type with a specific instance name.
transient-variable	Declares global variable with the lifetime of the running composition. When starting the composition, the variable is initialised to the init-value. If the type of the variable is text, the init-value can be set; otherwise, the initial value is empty.
type	The type-string is either a meme-type or a pon-type.("pon" is a JSON-like notation used in PalCom)
name-def	Defines a name used in the composition.
name-use	Uses a name defined inside or outside (e.g. message parameters) the composition.
text-literal	A text literal.
access	Access through a simple or a qualified name.
simple-access	Access something by name.
qualified-access	Access something qualified by another access.
device-access, service-access	Access to a device or service by the name defined in the composition.
message-access	Access to a message on a service.

synthesized-service	Specifies a synthesized service and giving names to it.
command-def, out-command-def, in-command-def	Defines the name of a message. Defines whether it is an in or an out-message.
parameter	Defines a parameter of a message on a synthesized service.
script	Defines the scripting part of the composition. Contains a list of outer when-dos that match commands. When a when-do matches it creates a new reaction.
when-do	Listens for a command that matches the condition. When it matches, it executes the branch.
command-condition	A condition waiting for a given message from a specific service. The message can either be a message sent to a synthesised service specified by the composition or a message sent from a connected oblivious service.
service-connected, service-disconnected	A condition waiting for a service to connect or disconnect. The service is specified by the access.
branch	Contains a list of actions.
action	A blocking or non-blocking action.
local-variable	Defines a local variable accessible in current and nested branches. Optional to have initial value. This is a non-blocking action.
assign	Assigns the value of the expr to the variable accessed by the access. The variable can be either local or global. This is a non-blocking action.
receive	Waits for a response specified by the access. This is a blocking action.
select	Blocks and waits for one of its when-dos to match.
parallel	Starts to execute all its branches. Blocks until all branches have finished.
finish-first	Starts to execute all its branches. Blocks until one branch has finished.

send-to-service	Access specifies what message is sent and to what service. The list of arguments is evaluated and embedded in the message.
arg	The access is the name of the argument. The expr is evaluated to the value of the argument.
send-from-synthesized	Send a message from a synthesized service. Access specifies what message on what synthesized service. The list of arguments is evaluated and embedded in the message. Send type is how to send the message.
send	Send a message out on a synthesized service to all that are connected to that service.
reply	Send a reply to the service that previously sent a request to the synthesized service that matched the condition of outer when-do.
expr	An expression.
message	References the latest received message.

A.3 Semantics

In this section, we describe parts of the semantics for COMPOS by showing the pseudo-code implementation of the interpreter. To only highlight the interesting features, we have omitted variables and service-connected conditions.

Epoch id and *Reaction id* are values attached to messages and used in the interpreter. Reaction ids are used to separate messages over the same connection for different reactions. The epoch id identifies each incarnation of a reaction, so that replies from already aborted reactions are ignored. In section A.3.1 we illustrate the use for reaction id and epoch id.

In addition to the AST nodes that hold the static code structure, we have two notable run-time data structures in the pseudo-code implementation, *Message* and *Reaction*. Below we have two tables describing the fields in these:

Message

Field	*Description*
connection	connection the message is sent over
toService	service the message is sent to
reactionId	reaction id of the sending service
epochId	epoch id of the sending service
matched	**TRUE** if message already matched an action, **FALSE** by default

Reaction

Field	*Description*
connection	connection to the service initiating the reaction
reactionId	the reaction id
epochId	the epoch id of the reaction
remoteReactionId	the reaction id in the message initiating the reaction
remoteEpochId	the epoch id in the message initiating the reaction
currentAction	pointer to the current action in the reaction
childReactions	a list of reactions, used in *parallel* and *finish-first*

To make the code easier to read, we have marked different types of tokens depending on their role: global variable, local variable, *AST node variable*, *AST node type*, state-changing procedure, and PROCEDURE DEFINED IN PSEUDO-CODE .

Algorithm 1 Pseudo-code for the event loop handling incoming messages

```
messageQueue                                  ▷ queue of messages arriving to the composition
composition                                   ▷ the AST of the composition
reactions := []                               ▷ list of all started reactions
epochId := 0                                  ▷ counter used to create new epoch id
reactionId := 0                               ▷ counter used to create new reaction id
loop
    message = messageQueue.waitForNextMessage()
    if composition.MATCHSPONTANEOUS(message) then
                                              ▷ new spontaneous incomming message
        reactionSpec := composition.findMatch(message)
        reaction := FINDREACTION(message)
        if reaction = NULL then               ▷ new reaction
            reaction := new Reaction()
            reaction.currentAction := reactionSpec.startAction()
            reaction.connection := message.connection
            reaction.reactionId := reactionId
            reaction.remoteReactionId := message.reactionId
            reaction.epochId := epochId
            reaction.remoteEpochId := message.epochId
            reactions.add(reaction)
            reactionId := reactionId + 1
        else                                  ▷ abort ongoing reaction
            reaction.epochId := epochId
            reaction.remoteEpochId := message.epochId
            reaction.removeChildReactions()
            reaction.currentAction := reactionSpec.startAction()
                         ▷ reaction id is reused in order to propagate the abort, see A.3.1
        end if
        RUNUNTILBLOCK(reaction, message)
        epochId := epochId + 1
    else
        for reaction in reactions do
                    ▷ There will be at most one reaction that matches (currently not checked)
            RUNUNTILBLOCK(reaction, message)
        end for
    end if
end loop
```

Algorithm 2 Pseudo-code for FINDREACTION, RUNUNTILBLOCK, and MATCH

```
procedure FINDREACTION(message)
    for reaction in reactions do
        if message.connection = reaction.connection
            ∧ message.reactionId = reaction.remoteReactionId then
            return reaction
        end if
    end for
    return NULL
end procedure

procedure RUNUNTILBLOCK(reaction, message)
    while reaction.currentAction.PERFORM(reaction, message) do
    end while
end procedure

procedure (receive ∨ message-condition).MATCHRESPONSE(reaction, message)
    return (message.name = this.message-access.messageName()
          ∧ message.connection = this.message-access.service().connection
          ∧ message.epochId = reaction.epochId ∧¬ message.matched )
end procedure

procedure Composition.MATCHSPONTANEOUS(message)
    for condition in this.allOuterMessageConditions() do
        access := condition.message-access
        if message.name = access.messageName()
            ∧ ( message.connection = access.service().connection
            ∨ message.toService = access.synthesizedService()) then
            return TRUE
        end if
    end for
    return FALSE
end procedure
```

Algorithm 3 Pseudo-code for PERFORM

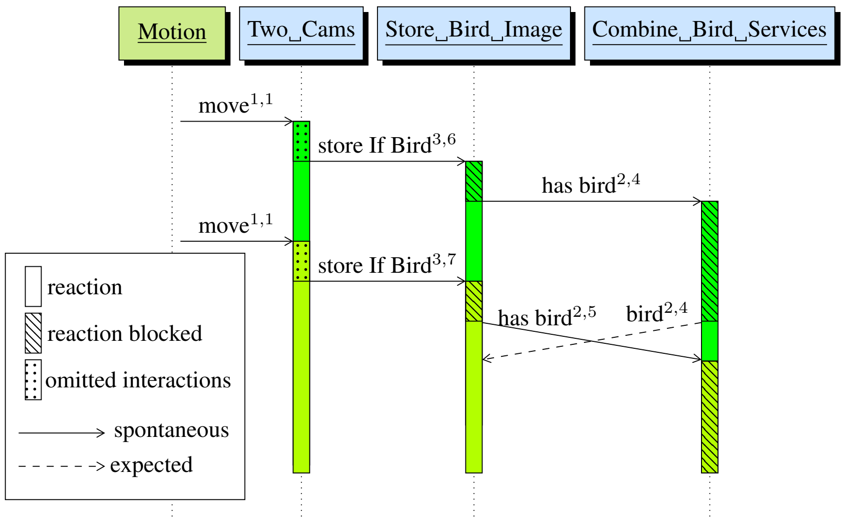
```
abstract procedure action.PERFORM(reaction, message)
                                        ▷ return TRUE to continue to next action
end procedure

procedure receive.PERFORM(reaction, message)
    if this.MATCHRESPONSE(reaction, message) then
        message.matched = TRUE
        reaction.currentAction = this.nextAction()
        return TRUE
    end if
    return FALSE
end procedure

procedure select.PERFORM(reaction, message)
    for whenDo in this.whenDos do
        if whenDo.condition.MATCHRESPONSE(reaction, message) then
            message.matched = TRUE
            reaction.currentAction = whenDo.startAction()
            return TRUE
        end if
    end for
    return FALSE
end procedure

procedure send-to-service.PERFORM(reaction, message)
    send(this.findService().connection, this.messageName(), this.args(. . . ),
            reaction.reactionId, reaction.epochId)
    reaction.currentAction = this.nextAction()
    return TRUE
end procedure

procedure send-from-synthesized.PERFORM(reaction, message)
    if this.send-type is send then
        for service in this.findSynthesizedService().connectedServices() do
            send(service.connection, this.messageName(), this.args(. . . ),
                    reaction.reactionId, reaction.epochId)
        end for
    else if this.send-type is reply then
        send(reaction.connection, this.messageName(), this.args(. . . ),
                reaction.remoteReactionId, reactionId.remoteEpochId)
    end if
    reaction.currentAction = this.nextAction()
    return TRUE
end procedure
```

Algorithm 4 Pseudo-code for PERFORM continue

```
procedure parallel.PERFORM(reaction, message)
    if ¬ reaction.hasChildReaction() then
        ▷ This branch is taken the first time performed is called on this node for a epoch id.
        for branch in this.branches do
            childReaction := new Reaction()
            childReaction.currentAction := branch.startAction()
            childReaction.connection := message.connection
            childReaction.reactionId := reaction.reactionId
            childReaction.remoteReactionId := reaction.remoteReactionId
            childReaction.epochId := reaction.epochId
            childReaction.remoteEpochId := reaction.remoteEpochId
            reaction.addChildReaction(childReaction)
        end for
    end if
    for childReaction in reaction.childReactions do
        RUNUNTILBLOCK(childReaction, message)
    end for
    for childReaction in reaction.childReactions do
        if ¬ childReaction.isFinish() then
            return FALSE
        end if
    end for
    reaction.removeChildReactions()
    reaction.currentAction = this.nextAction()
    return TRUE
end procedure

procedure finish-first.PERFORM(reaction, message)
    ...                                          ▷ The same beginning as Parallel
    for childReaction in reaction.childReactions do
        if childReaction.isFinish() then
            reaction.removeChildReactions()
            reaction.currentAction = this.nextAction()
            return TRUE
        end if
    end for
    return FALSE
end procedure
```

A.3.1 Example for use of reaction id and epoch id

To highlight the use of the reaction id and epoch id, we have an example on a message sequence in the refactored two camera bird watcher scenario from section 4. The message sequence is shown in a sequence diagram below. To make the example clearer, we have omitted some of the interactions. We have annotated every message in the sequence diagram with two numbers, the first number is the reaction id and the second number is the epoch id. The first that happens in the sequence is that the Motion Sensor sends a move message. The Two␣Cams composition then takes a photo, omitted in the diagram, and sends it to Store␣Bird␣Image with reaction id 3 and epoch id 6. Store␣Bird␣Image then sends *bird* to the Combine␣Bird␣Services with reaction id 2 and epoch id 4. In this example, the motion sensor then gets triggered again and aborts Two␣Cams that starts a new reaction with the same reaction id and a new epoch id. We need to use the same reaction id, 3, when we create a new reaction to be able to abort depending reactions, in this case, the depending reaction of Store␣Bird␣Image. Store␣Bird␣Image aborts its reaction and starts a new one with the same reaction id and epoch id 5. Before Combine␣Bird␣Services receives the new *has bird* message, it replies *bird* with epoch Id 4. The only way Store␣Bird␣Image differentiates this old reply from a reply for the message just sent is by looking at the epoch id and sees that the reply is from epoch 4 and now it is epoch 5. The reply is then ignored, since it is from an older aborted reaction.



The reply bird^{2,4} is ignored since it arrives when the epoch has changed to 5

Appendix B Full examples

B.1 Bird watcher composition

Full code for the bird watcher composition.

```
BirdWatcher version: 1
  bindings:
    motion_sensor = MotionSensor
    camera = Camera
    computer = Computer
    motion_service = Motion_Service:1 on motion_sensor
    camera_service = Camera_Service:1 on camera
    bird_service = Bird_Service:1 on computer
    storage_service = Storage_Service:1 on computer
  synthesized services:
  script:
    when
      motion_service.move
    do
      send to camera_service.take_photo
      receive camera_service.photo
      var photo = message
      send to bird_service.has_bird(
        image = photo.img
      )
      select
        when
          bird_service.bird
        do
          send to storage_service.store_image(
            image = photo.img
          )
        when
          bird_service.not_bird
        do
```

B.2 Extended example

B.2.1 Two␣Cams

Full code for the Two␣Cams composition before refactoring.

```
Two_Cams version: 1
  bindings:
    motion_sensor = MotionSensor
    camera1 = Camera1
    camera2 = Camera2
    computer = Computer
    motion_service = Motion_Service:1 on motion_sensor
    camera_service_1 = Camera_Service:1 on camera1
    camera_service_2 = Camera_Service:1 on camera2
    bird_service_1 = Bird_Service:1 on computer
    bird_service_2 = Bird_Service:1 on computer
    storage_service_1 = Storage_Service:1 on computer
    storage_service_2 = Storage_Service:1 on computer
  synthesized services:
  script:
    when
      motion_service.move
    do
      parallel
        send to camera_service_1.take_photo
        receive camera_service_1.photo
        var photo = message
        send to bird_service_1.has_bird(
          image = photo.img
        )
        select
          when
            bird_service_1.bird
          do
            send to storage_service_1.store_image(
              image = photo.img
            )
          when
            bird_service_1.not_bird
          do
      with
        send to camera_service_2.take_photo
        receive camera_service_2.photo
        var photo = message
        send to bird_service_2.has_bird(
          image = photo.img
        )
        select
          when
            bird_service_2.bird
    do
      send to storage_service_2.store_image(
        image = photo.img
      )
    when
      bird_service_2.not_bird
    do
end
```

B.2.2 Combine␣Bird␣Services

Full code for the Combine␣Bird␣Services composition.

```
Combine_Bird_Services version: 1
  bindings:
    server = Server
    computer = Computer
    remote_bird = Bird_Service:1 on server
    local_bird = Bird_Service:1 on computer
  synthesized services:
    synthesized service Bird:
      in: has_bird
        image[image/jpeg]
      out: not_bird
      out: bird
  script:
    when
      Bird.has_bird
    do
      var has_bird_cmd = message
      send to remote_bird.has_bird(
        image = has_bird_cmd.image
      )
      send to local_bird.has_bird(
        image = has_bird_cmd.image
      )
      finish first
        receive local_bird.bird
        reply from Bird.bird
      or
        receive remote_bird.bird
        reply from Bird.bird
      or
        parallel
          receive local_bird.not_bird
        with
          receive remote_bird.not_bird
        end
        reply from Bird.not_bird
      end
```

B.3 Adding Store␣If␣Bird

B.3.1 Two␣Cams using Store␣If␣Bird

Full code for the Two␣Cams composition refactored to use Store␣If␣Bird.

```
Two_Cams version: 2
bindings:
  motion_sensor = MotionSensor
  camera1 = Camera1
  camera2 = Camera2
  computer = Computer
  motion_service = Motion_Service:1 on motion_sensor
  camera_service_1 = Camera_Service:1 on camera1
  camera_service_2 = Camera_Service:1 on camera2
  store_if_bird_1 = Store_If_Bird:1 on computer
  store_if_bird_2 = Store_If_Bird:1 on computer
synthesized services:
script:
  when
    motion_service.move
  do
    parallel
      send to camera_service_1.take_photo
      receive camera_service_1.photo
      var photo = message
      send to store_if_bird_1.store_if_bird(
        image = photo.img
      )
    with
      send to camera_service_2.take_photo
      receive camera_service_2.photo
      var photo = message
      send to store_if_bird_2.store_if_bird(
        image = photo.img
      )
    end
```

B.3.2 Store␣Bird␣Image

Full code for the Store␣Bird␣Image providing the synthezised service Store␣If␣Bird.

```
Store_Bird_Image version: 1
bindings:
  computer = Computer
  bird_service = Bird_Service:1 on computer
  storage_service = Storage_Service:1 on computer
synthesized services:
  synthesized service [Store_If_Bird]:
    in: store_if_bird
      image[image/jpeg]
script:
  when
    Store_If_Bird.store_if_bird
  do
    var msg = message
    send to bird_service.has_bird(
      image = msg.image
    )
    select
      when
        bird_service.bird
      do
        send to storage_service.store_image(
          image = msg.image
        )
      when
        bird_service.not_bird
      do
```

B.4 Latch

Full code for the latch composition.

```
Latch version: 1
  bindings:
    motion_sensor = MotionSensor
    camera = Camera
    computer = Computer
    motion_service = Motion_Service:1 on motion_sensor
    camera_service = Camera_Service:1 on camera
    bird_service = Bird_Service:1 on computer
    storage_service = Storage_Service:1 on computer
    latch_service = Latch_Service:1 on computer
  synthesized services:
  script:
    when
      latch_service.signal
    do
      send to camera_service.take_photo
      receive camera_service.photo
      var photo = message
      send to bird_service.has_bird(
        image = photo.img
      )
      select
        when
          bird_service.bird
        do
          send to storage_service.store_image(
            image = photo.img
          )
        when
          bird_service.not_bird
        do
      send to latch_service.reset
    when
      motion_service.move
    do
      send to latch_service.signal
```