ComPOS – a Domain-Specific Language for Composing Internet-of-Things Systems

Alfred Åkesson

Doctoral thesis, 2021

Department of Computer Science
Lund University

ISBN: 978-91-7895-906-8 (printed version)
ISBN: 978-91-7895-905-1 (electronic version)
ISSN: 1404-1219
Dissertation 66, 2021
LU-CS-DISS: 2021-02
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, 2021

© *2021 Alfred Åkesson*

Abstract

Internet-of-Things (IoT) systems consist of spatially distributed interacting devices. In contrast to desktop applications, IoT systems are always running and need to deal with unresponsive devices and weak connectivity. In this thesis, we propose techniques for simplifying the development of such systems. The work addresses IoT systems organised as reusable services connected by compositions. We propose to program such compositions using stateful reactions that mediate messages. To this end, we have designed a domain-specific language (DSL), called COMPOS. To help systems operate partly in cases of weak connectivity, we propose that COMPOS aborts older reactions when newer messages arrive. We evaluate our DSL in home-automation and e-health scenarios.

Understanding IoT systems can be hard, and different analyses can help explain how they work. To support analysis, we propose a conceptual runtime model based on relational reference attribute grammars. We demonstrate the approach by formulating and implementing a Device Dependency Analysis (DDA). The DDA finds sets of devices needed for given parts of the system to work.

The COMPOS editor supports live programming to allow development while the system is running. We propose a methodology for live COMPOS programming which divides the development into three, iteratively applied, phases: finding services (explore), composing services (assemble), and abstracting compositions as new services (expose).

When developing a DSL, it takes substantial effort to specify the syntax and semantics, to build tools like editors, and to integrate with the environment (in this case the underlying middleware). To reduce the effort needed to experiment with COMPOS, we have created a tool called JATTE. JATTE is a generic projectional editor that developers can tune using attribute grammars. We used JATTE to implement the COMPOS editor.

Acknowledgements

First of all, I like 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. I also 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 your insight about how PALCOM is used. Thanks, Jesper Öqvist, my go-to guy for JASTADD problems. I like to thank all the other members of my research group, Software Development Environment. Also, I like to thank all the members of the computer science department for all fika and support.

Another thanks goes to my collaborators in Dresden, Rene Schöne and Johannes Mey. Unfortunately, I could not visit you because of the Covid-19 pandemic.

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.

This work was also in part supported by the Swedish Foundation for Strategic Research (SSF), grant RIT17-0035 SMARTY. Thanks, SSF.

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 men även 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.

Contributions by the author

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

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

Paper I 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

Paper II 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

Included as a revisited and extended version with the title: "COMPOS: Composing Systems of Services".

Paper III Alfred Åkesson, Görel Hedin, Niklas Fors, Rene Schöne, and Johannes Mey. “Runtime Modeling and Analysis of IoT Systems”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’20. Virtual Event, Canada: Association for Computing Machinery, 2020

Paper IV 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, 2017, pp. 7–12

The table below shows the contributions of the author.

Paper	Concept	Implementation	Evaluation	Writing
I (2018)	◑	N/A	◕	◑
II (2019-2021)	◕	●	●	◕
III (2020)	◕	●	●	◕
IV (2017)	◑	●	●	◑

● Lead and did almost all the work

◕ Lead and did a majority of the work

◑ Contributed to a majority of the work

◔ Contributed to a minority of the work

Concept Coming up with the ideas of the paper

Implementation Implementing the software described in the paper

Evaluation Conducting the evaluation described in the paper

Writing Drafting and editing the paper

Worth noting is that the conceptualization often was done collaboratively by all the authors during meetings.

COMPOS – ett Programmeringsspråk för att Koppla Ihop Smarta Prylar

Datorer som kan kopplas upp mot internet blir allt billigare och mindre. Detta gör att de kan integreras i produkter för att skapa så kallade smarta prylar. Exempel på prylar som kan vara smarta är lampor, termostater, vågar, blodtrycksmätare, lås etc. Denna trend med uppkopplade smarta prylar kallas sakernas internet. Sakernas internet kan utnyttjas på olika områden, så som i hemmet, sjukvården, industrin eller inom jordbruket. Genom att koppla ihop flera smarta prylar skapas ett så kallat sakernas-internetsystem. I dessa system samarbetar olika prylar för att bli till större nytta än varje pryl för sig själv. Ett exempel på ett sakernas-internetsystem är ett smart lås kopplat till smarta lampor, så att när du låser upp ditt hem, tänds dina lampor.

I denna avhandling vill vi göra det enklare att utveckla sakernas-internetsystem. I de system vi undersöker kan prylarna berätta vilka meddelande de kan skicka och ta emot. Sedan finns det små datorprogram, kallade kompositioner, som kopplar ihop prylarna. För att specificera kompositioner, har vi i avhandlingen skapat och utvärderat ett programmeringsspråk med detta som enda syfte. Vårt programmeringsspråk heter COMPOS. Genom att göra ett programmeringsspråk med ett enda syfte kan vi bortse från många funktioner som vanligen finns i mer generella programmeringsspråk och på så sätt göra programmen enklare att förstå, använda och analysera.

Koden nedan visar en komposition skriven i COMPOS som tänder två lampor om någon låser upp ytterdörren. Kompositionen kan köras på vilken dator som helst, t.ex. det smarta låset eller Wi-Fi routern. Den första raden instruerar datorn att vänta på att låset ska meddela att någon har låst upp dörren. Rad 2-3 instruerar datorn att skicka meddelande för att tända lampan i hallen och i köket. Att rad 2-3 är indragna betyder att de utförs först efter det på rad 1 har hänt.

```
1 when receive låst upp from ytterdörrens lås do
2     send tänd to lampa hallen
3     send tänd to lampa köket
```

En skillnad mellan ett system på en enda dator och ett sakernas-internetsystem, är att de senare måste hantera att prylar tappar kontakten och kanske senare får kontakt igen. Vi har designat COMPOS så att när prylar tappar kontakten, blir det en begränsad påverkan på resten av systemet. T.ex. om lampan i hallen tappar kontakten med systemet, ska lampan i köket fortfarande tändas när man låser upp sitt smarta lås.

Sakernas-internetsystem blir lätt komplexa med prylar som skickar meddelanden kors och tvärs. Genom att ha kompositioner som beskriver hur meddelandena flödar i systemet kan vi analysera dem för att få inblick i hur ett system fungerar. I denna avhandling har vi visat hur körande system kan modelleras och analyseras på en hög nivå. Som exempel har vi skapat en analys som räknar ut

vilka delar av systemet som fungerar när olika prylar tappar kontakten. Detta kan till exempel vara användbart om man vill försäkra sig om att lampan i köket tänds, även när lampan i hallen har tappat kontakten.

För att vidare underlätta utvecklingen av sakernas-internetsystem, har vi integrerat COMPOS i en programmeringsmiljö som tillåter användaren att se vilka prylar som är uppkopplade. Användaren kan sedan "dra" en uppkopplad pryl till COMPOS-editorn för att använda prylen i en COMPOS-komposition. Som en del i att utveckla programmeringsmiljön, har vi skapat och utvärderat ett editor-ramverk, som heter JATTE. JATTE kan användas till olika programmeringsspråk och integreras i olika programmeringsmiljöer.

För att utvärdera COMPOS, har vi tagit ett sakernas-internetsystem för hemsjukvård av njursviktpatienter, skrivit med ett annat kompositionsspråk, och återskapat kompositionerna med COMPOS. Jämnfört med ursprungliga kompositionsspråk är meddelandeflödet i COMPOS tydligare. Fördelarna med COMPOS är att man enklare kan programmera komplicerade meddelandeflöden och skapa analyser. Med denna forskning hoppas vi kunna bidra till att göra det enklare och och snabbare att utveckla sakernas-internetsystem.

CONTENTS

1 Introduction 1
2 IoT Middleware 2
3 Programming-Language Techniques 7
4 Domain-Specific Languages 14
5 Research Approach 15
6 Contributions 16
7 Conclusions and Future Work 21
References 21

I Live Programming of Internet of Things in PALCOM **27**
1 Introduction 27
2 The PalCom Middleware Toolkit 28
3 Live Programming in PalCom 29
4 Example: Photo Booth 30
5 Related Work 36
6 Conclusions 37
7 Acknowledgements 38
References 38

II COMPOS: Composing Systems of Services **41**
1 Introduction 41
2 IoT Architecture 43
3 Motivating Example 47
4 The COMPOS Language 49
5 Composing Scenarios 55
6 The Abort Strategy 60
7 Adapting Semantics with Strategy Services 71
8 Evaluation 77

9 Related Work 85
10 Conclusions and Future Work 88
References 89
Appendix A A COMPOS Specification 93
Appendix B Home Automation Scenarios 100

III Runtime Modeling and Analysis of IoT Systems 105
1 Introduction 105
2 Basic Runtime Model 107
3 Running example 109
4 The Full System Model 110
5 Device Dependency Analysis 112
6 Related Work 116
7 Conclusion 116
8 Acknowledgements 117
References 117

IV JATTE: A Tunable Tree Editor for Integrated DSLs 119
1 Introduction 119
2 Background 120
3 Default Tree Editor 121
4 Customizing the Editor 122
5 Case Study: IoT Language 126
6 Implementation 130
7 Related Work 130
8 Conclusion 131
References 131

1 Introduction

Currently, we see a trend of cheaper computers with better connectivity being embedded into devices. This trend is commonly referred to as the Internet of Things (IoT) [AIM10]. These communicating devices enable new types of systems to emerge called *IoT systems*, containing multiple connected devices. An example of an IoT system 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 an IoT system is in home automation, where the colour of a light indicates the home's energy consumption [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 individual devices may shut down or disconnect.

Another challenge is to understand what happens in the system during runtime and 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 try to simplify the process of building them so that end-users, who typically have no programming skills, can do so themselves [Tet+15].

How can we simplify development of IoT systems? This is the main question that we strive to answer with this research. We take steps to answer this question by proposing a domain-specific language (DSL) (Paper II), an analysis of IoT systems (Paper III), a development environment with support for live programming (Paper I), and a tool for experimenting with the DSL and the development environment (Paper IV).

Our proposed DSL is a stateful composition language called COMPOS and is designed to handle weak connectivity (Paper II). To help understand IoT systems, we propose the *Device Dependency Analysis (DDA)* to find sets of devices that need to be connected, in order for a specified message to be sent (Paper III). This paper also proposes a runtime model for IoT systems to enable analysis like the DDA. Our development environment supports live programming [Tan90; Tan13] to allow users to explore and evolve always-running IoT systems (Paper I). 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 IV).

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] (using the JASTADD

meta-compiler [HM03]). To specify the runtime model and the DDA, we used relational reference attribute grammars [Sch+19].

The rest of this chapter includes three sections of background (sections 2-4). The first background section describes IoT middleware and PALCOM (Section 2). The next section describes some of the programming-language techniques used in this thesis (Section 3). The last background section describes domain-specific languages and their benefits (Section 4).

Then follows a description of how we did the research in this thesis (Section 5) and a summary of the contributions of the thesis (Section 6). Finally, we present conclusions and future work (Section 7).

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 on and expands the PALCOM middleware toolkit. In this section, we will first describe PALCOM and then look at different classifications of IoT middleware to classify PALCOM. Lastly, we compare PALCOM to the ZigBee standard which is commonly used in home automation.

2.1 PALCOM

PALCOM [SF09; SF+09] is a middleware toolkit for building IoT systems. A PALCOM system consists of *services* connected by *compositions*. When a composition connects to a service, it receives the messages the service sends. The composition can then adapt these messages and send them to other services. A service has an interface describing the messages it can send and receive.

Services and compositions run on *devices*. A device is a running middleware instance with a globally unique identifier called a *device id*. Devices automatically discover each other over the network, using the PALCOM discovery protocol. The discovery protocol also communicates what services and compositions a device hosts.

One kind of services are the *native* services; they perform computations and interact with the physical environment. For example, a native service can control a light, compute a control signal, or store an image. To enable this wide range of functionality, native services are implemented in general-purpose programming languages, like Java or C.

Compositions are limited to mediating and adapting messages, and a DSL is used for their specification. To provide an abstraction mechanism, compositions can combine multiple services' functionality into one *synthesized* service. For example, a synthesized service can provide a single interface for controlling multiple light services.

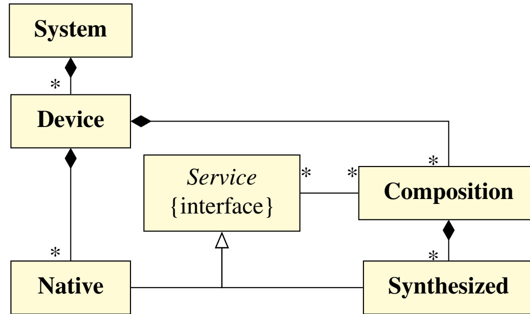


Figure 1: Conceptual model for PALCOM systems.

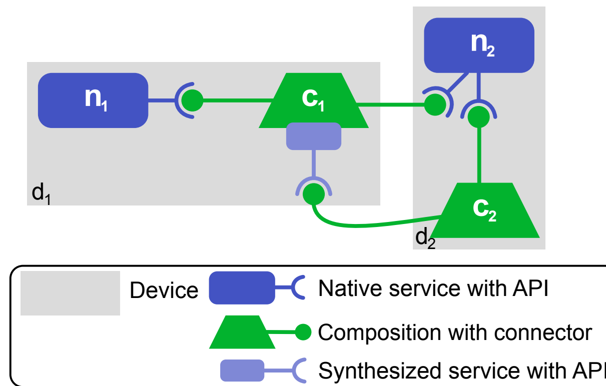


Figure 2: An instance of the conceptual model in Figure 1.

Figure 1 shows a conceptual model for PALCOM systems. A PALCOM system consists of devices that host native services and compositions. A composition connects to zero or more services. A service can be either native or synthesized, the latter being part of a composition. Each service has an interface of incoming and outgoing messages. Messages can have parameters to transfer data between services. Figure 2 shows an instance of the conceptual model with two devices d_1 and d_2. Device d_1 hosts the native service n_1 and the composition c_1. Composition c_1 connects to n_1 and n_2 to provide a synthesized service. The native service n_2 and composition c_2 are hosted on d_2, and c_2 connects to n_2 and the synthesized service provided by c_1.

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 on the right is an editor for creating and editing compositions. In our work, we have integrated a new editor for our new composition language, COMPOS, into the

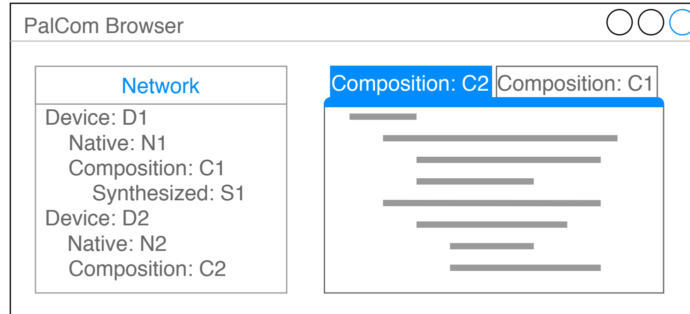


Figure 3: A sketch of the PALCOM Browser with the discovered devices and services on the left and the composition editor on the right.

PALCOM Browser.

2.2 IoT-middleware Classification

To give a better intuition about PALCOM, we will in this section describe three different ways to classify IoT-middleware and try to fit PALCOM into these classifications. We will describe the following classifications:

- Service-based, Cloud-based, or Actor-based [Ngu+17] (Section 2.2.1)
- Communication Interaction Models [Eug+03] (Section 2.2.2)
- Orchestration and Choreography [Erl05] (Section 2.2.3)

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

Ngu, Gutierrez, Metsis, Nepal, and Sheng [Ngu+17] propose to classify IoT middlewares into the following three categories:

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. An example of a cloud-based middleware is Google Fit, where device manufacturers use an API to upload data from their fitness trackers to the Google Fit cloud. Google Fit allows users to have all their fitness data in one place and free device manufacturers from the need to have a fitness cloud.

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 communication between the IoT devices; instead, the devices communicate

with the services. Compared to the Cloud-based middleware, the developer deploys the middleware on the server and can also run custom code on it. An example of a service-based middleware is Global Sensor Network, which allows developers to connect data from different devices to their own customized instance of the Global Sensor Network.

Actor-based Actors are programs that can be added dynamically to a device running the actor middleware. All devices in a system run the middleware. Every device that fulfills 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. Calvin is an actor-based IoT middleware that allows actors to move between devices on the network's edge to reduce latency.

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, thus creating an open system.

Paper I calls PALCOM a service-based middleware. However, we do not think that PALCOM fits the definition of service-based middleware given by Ngu, Gutierrez, Metsis, Nepal, and Sheng [Ngu+17]. When we say that PALCOM is service-based, we allude to the fact that PALCOM has services as part of the architecture.

2.2.2 Decoupling Properties

Eugster, Felber, Guerraoui, and Kermarrec [Eug+03] suggest three decoupling properties for the interaction between a sender and a receiver. They use these decoupling properties to classify different interaction paradigms, e.g., message passing and publish/subscribe. The decoupling properties are:

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 to actively wait for messages to arrive.

Table 1 shows decoupling properties of PALCOM and the following paradigms: *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

	Decoupling properties			
	Space	Time	Synchronization	Interface
Message passing	No	No	Only sender	N/A
RPC	No	No	Only sender	N/A
Tuple space	Yes	Yes	Only sender	No
Pub/Sub	Yes	Yes	Yes	No
PALCOM services	Yes	Yes	Yes	Yes

Table 1: Different interaction paradigms and their decoupling properties. Grey parts are from [Eug+03]. *Only sender* means the sender of message does not need to block, but the receiver of messages has to.

program distributed systems because there is no difference between calling a remote procedure and a local one [Eug+03]. *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 [Eug+03].

In PALCOM, two services interact with each other through a composition. PALCOM is space decoupled because services do not know to whom they talk. For time decoupling, we need a third party to store the message when neither the sending service or receiving service is connected. A composition could act as such a third party by using reliable connections and long time outs. Time decoupling could also be accomplished by adding a caching service. In PALCOM, messages are sent and received asynchronously, making PALCOM synchronization decoupled.

We also suggest another type of decoupling called *interface decoupling*.

Interface decoupling The interacting parties do not need to have the same interface, i.e., no pre-agreed message name or format.

Two communicating services in PALCOM are interface decoupled because the composition between them can adapt the sender's message to fit the receiver's expectations. Tuple spaces are not interface decoupled because the sender and the receiver must agree on the tuple structure. Similarly, Pub/Sub is not interface decoupled because the sender and receiver must agree on the message format and maybe topic. Because messages passing and RPC are space coupled, thus already know each other, interface decoupling is not applicable here.

2.2.3 Orchestration and Choreography

Two approaches for service composition are orchestration and choreography [Erl05; Pel03]. 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 the different roles services can play in a choreography.

PALCOM uses a combined approach. On one hand, each composition acts as the conductor, orchestrating services. On the other hand, devices can communicate peer-to-peer and since compositions may provide synthesized services, the system becomes distributed over different devices without a central conductor. This corresponds, at a system level, to a choreography without any description. In Figure 2, earlier shown, we see each composition as orchestrating services; at the same time, there is no central conductor in the system.

2.3 Home Automation with Zigbee

Zigbee is a popular standard in home automation, powering smart-light solutions from well-known brands such as Philips and IKEA. The Zigbee standard defines a full communication stack, based on low-bandwidth wireless communication, for building IoT system [HBE11]. Zigbee provides interoperability between IoT devices through the ZigBee Cluster Library (ZCL) [Gis08]. Devices interoperate through roles in predefined scenarios, e.g. a light bulb and a switch, in a home automation scenario.

PALCOM does not have predefined scenarios; instead, self-describing services can interoperate with other services using compositions. For the user, the PALCOM approach is more flexible by not being limited to predefined scenarios.

3 Programming-Language Techniques

In this thesis, we use techniques that emerged in the programming-language community. Understanding these techniques helps with understanding the thesis. This section gives background to some of the programming-language techniques we use.

3.1 Abstract Syntax Tree

When developing a program, it must adhere to the programming language's rules for writing the source code text, i.e., the *concrete syntax*. The program can then be parsed into an *abstract syntax tree (AST)*. The AST represents the input program inside a compiler. The nodes in the AST represent different parts of the program. For example, the program

```
                              class B extends A {
                                C c;
B:A ::= C D? F* <t>;            Opt<D> d;
                                List<F> fs;
                                String t;
                              }
```

Figure 4: The figure shows a JASTADD abstract grammar rule (left) and the corresponding (simplified) generated Java code (right).

```
let int b = 0 in 1 + b
```

can be represented by the AST in Figure 5a.

Like how the concrete syntax rules govern source code text, the *abstract syntax* rules govern AST structure. For specifying the abstract syntax in this thesis, we use the abstract grammar language in the meta-compilation system JASTADD. JASTADD generates a Java class hierarchy from the abstract grammar. For example, the abstract grammar rule:

```
B:A ::= C D? F* <t>;
```

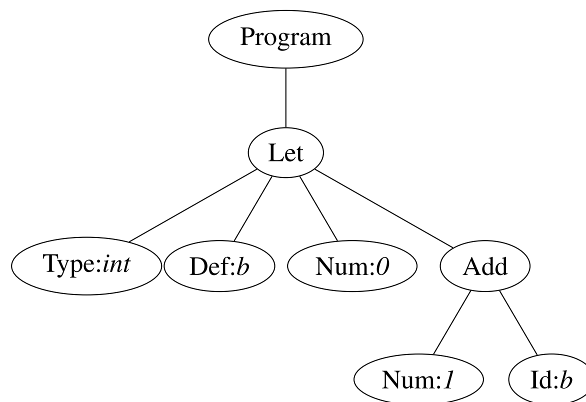
generates the class `B` that inherits from `A` and contains a child of type `C`, an optional `D`, a list of `F`s and a string token `t` (see Figure 4). The abstract grammar used for the AST in Figure 5a is shown in Figure 5b.

3.2 Reference Attribute Grammars

We use reference attribute grammars in papers II, III, and IV. Reference attribute grammars extend *attribute grammars* [Knu68]. In attribute grammars, an *attribute* is a computed property of an AST node. An AST node class declares an attribute, and one or more *equations* define the value. The right-hand side of an equation is an expression that can depend on other attributes. The left-hand side defines what attribute the right-hand expresses and the expression's evaluation context.

In its simplest form, there are two kinds of attributes: synthesized(↑) and inherited(↓). A synthesized attribute is declared on a node class and is like a virtual method: the defining equations of a synthesized attribute are in the class or subclasses. An inherited attribute is also declared on a node class, but its defining equations are in ancestor nodes. 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. It also allows information to flow across the AST via reference attribute edges and not only along the AST tree structure.



(a) Abstract Syntax Tree for `let int b = 0 in 1 + b`

```
1  Program ::= Expr;
2  Type ::= <type>;
3  Def ::= <def>;
4  abstract Expr;
5  Id:Expr ::= <id>;
6  Num:Expr ::= <num:int>;
7  Add:Expr ::= left:Expr right:Expr;
8  Let:Expr ::= Type Def Num Expr;
```

(b) Example grammar for the language AST above

Figure 5: AST with its corresponding grammar

Listing 1: Specfication of the name analysis.

```
1  ↑ Id.def : Def
2  ↓ Id.lookup(id:String) : Def
3  ↓ Let.lookup(id:String) : Def
4  ↑ Def.localLookup(id:String) : Def
5  eq Id.def = lookup(this.id)
6  eq Program.Expr.lookup(id:String) = null
7  eq Let.Expr.lookup(id:String) = this.Def.define(id)
8    ? this.Def : lookup(id)
9  eq Def.define(id:String) = this.def == id
```

Listing 2: Specfication of the type analysis.

```
1  ↑ Id.type : String
2  ↓ Def.type : String
3  eq Id.type = this.def.type
4  eq Let.Def.type = this.Type.type
```

Listings 1 and 2 show the name and type analysis implemented using reference attribute grammars for the language in Figure 5b. In Listing 1, line 1-4 declare attributes (`def`, `lookup`, and `localLookup`), and lines 5-9 give the equations for these attributes. Line 1 declares the synthesized reference attribute ↑`def`, on an `Id` node, that refers to the definition, i.e., a `Def` node. The equation on line 5 defines the value of ↑`def` by calling the inherited attribute ↓`lookup` with the `id` token as the parameter. Because ↓`lookup` is an inherited attribute, the equation is defined in an ancestor node, in this case, the nearest ancestor node of type `Let` or `Program`. The equation for ↓`lookup` in the `Let` (line 7-8) checks if its `Def` defines the `id` using ↑`define` (line 9). If the `Def` defines the `id`, then the `Let` returns its `Def` otherwise it calls ↓`lookup` on the nearest `Let` or `Program` ancestor. As the base case for ↓`lookup`, `Program` (line 6) defines its equation to `null`.

The type analysis can leverage reference attributes by having the ↑`type` attribute of an `Id` follow its ↑`def` attribute to the corresponding `Def` node and find the type there (line 3 in Listing 2). Figure 6 shows the ↑`def` and ↑`type` attributes of an `Id` node for our running example.

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. Furthermore, the specification can be modularized using *aspect-oriented programming*.

Aspect-oriented programming is a mechanism for modularisation of cross-cutting concerns [Kic+97]. JASTADD supports aspect-oriented programming by

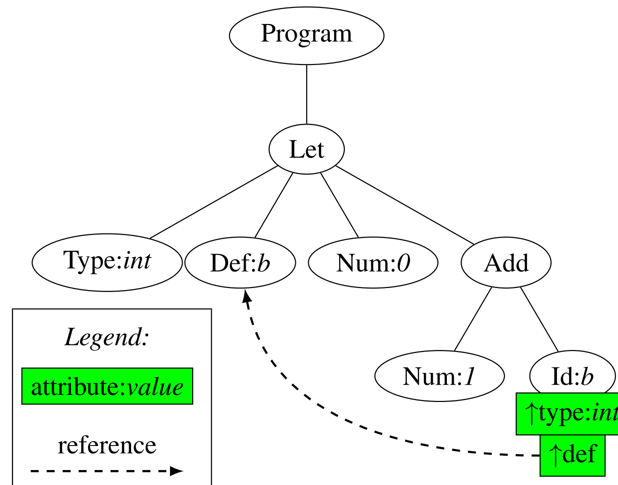


Figure 6: Illustration of ↑`def` and ↑`type` on the AST for
`let int b = 0 in 1 + b`.

allowing different members of a class to be defined separately in different aspects, like open classes in MultiJava and inter-type declarations in AspectJ [Cli+00; Kic+01]. These aspects can be used to separate different parts of a compiler implementation. Examples of different aspects are name analysis, type analysis, and interpretation. Aspects allow class members to be defined in aspect files, syntactically outside of their respective classes. The members are fields, methods, or attributes that JASTADD combines into Java AST classes.

JASTADD translates attributes into Java methods that, when called, locates and evaluates the defining equation. If the defining equation depends on other attributes, they are also evaluated. JASTADD can memorize attributes for efficiency: the first time it computes an attribute, the value is cached and used in subsequent calls.

As an example, JASTADD combines the abstract grammar in Figure 5b with the name and type analysis in listings 1 and 2 into AST classes containing methods for all the attributes (see Figure 7). The generated `Id` class will have attribute methods for both the ↑`def` and ↑`type` attributes.

Relational Reference Attribute Grammars

Relational RAGs extend the abstract syntax of RAGs with relations so that the structure to attribute can be a conceptual model rather than an AST only [Mey+20]. A conceptual model can model arbitrary relations between objects compared to an AST that only can model containment relations. For example, it is possible to model the many-to-many relation between services and compositions in Figure 1

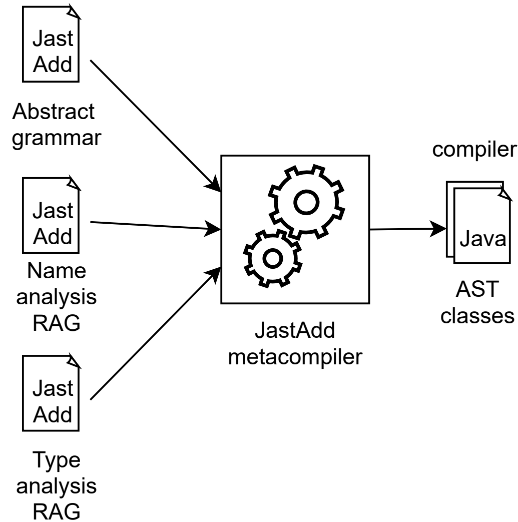


Figure 7: JASTADD constructs the compiler's core by combining the abstract grammar, name analysis and type analysis into Java classes.

using Relational RAGs. While it is possible to model non-containment relations with ordinary RAGs by using reference attributes resolved by name analysis, it is more straightforward with Relational RAGs.

To implement Relational RAGs, Mey et al. [Mey+20] added a preprocessor to JASTADD that translates non-containment relations to behave like reference attributes. Relational RAGs makes it possible to use attribute grammars on top of conceptual models.

In Paper III, we use Relational RAGs to model PALCOM IoT systems. We can then analyze PALCOM systems using attribute grammars and reuse attributes from the COMPOS specification in Paper II.

3.3 Program Analysis

In Paper III, we use program-analysis techniques to analyse IoT systems. This section gives an overview of static and dynamic analysis and why it is relevant for the thesis. This section also describes what a control-flow graph is and how we use it in Paper III.

Static vs Dynamic

Program analysis can be divided into static analysis and dynamic analysis, see for example Ernst [Ern03]. Static program analysis is about analysing the code

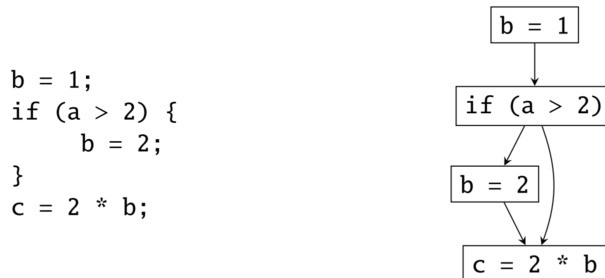


Figure 8: The figure shows a piece of code (left) and the corresponding control flow graph (right).

without running it. Dynamic analysis is about analysing a program by running it or during runtime, see Artho and Biere [AB05]. A static analysis considers all possible inputs of a program while dynamic analysis can realistically only consider a small subset of all inputs. Dynamic analysis is always precise for the given input.

It is also possible to combine static and dynamic analysis, which we do in Paper III in this thesis. In this paper, we build a model of the system at runtime (dynamic analysis) to find the connections between services and compositions. We then analyse the source code of compositions (static analysis) to infer how messages can flow. By combining static and dynamic analysis, we can consider all possible ways messages can flow for a particular running system.

Control-flow graph

In Paper III, we use a control-flow graph for the static part of our analysis. A control-flow graph (CFG) is a graph describing all the different paths through a piece of code. The nodes in the CFG represent actions in the language where each node has a set of successors, consisting of all the actions that can follow it [AP02]. An example of a CFG, with the corresponding source code, is shown in Figure 8.

In COMPOS, the CFG nodes are the AST nodes for the different actions in the language. We have implemented our CFG using RAGs, inspired by the approach described by Nilsson-Nyman, Ekman, Hedin, and Magnusson [NN+08].

3.4 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. The editing is done using operations for

adding, removing, moving and changing AST nodes. COMPOS scripts are edited using a projectional editor that visualizes the AST using textual notation. This editor is built using our meta tool JATTE, discussed in Paper IV. JATTE generates a projectional editor from a reference attribute grammar.

3.5 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 I 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 I, we argue that the PALCOM-development environment supports liveness between level 3 and level 4.

4 Domain-Specific Languages

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 [Hud96; 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, thus speeding up IoT-system development.

Validation and Verification A script written in COMPOS contains much semantic information that we use when analyzing IoT systems.

Productive Tooling Having COMPOS as a DSL allows us to create a custom editor using our meta tool JATTE. We have integrated the editor into the PALCOM Browser to support high-level domain-specific editing, e.g., allowing the user to 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 address them:

Evolution and Maintenance When experimenting with the COMPOS language, it is often hard to have backward compatibility. However, 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, and replacing these underlying frameworks would be a considerable investment. However, this is not unique for JASTADD and JATTE; it is always a considerable investment to change a DSL's underlying frameworks. COMPOS uses XML for serialisation, so creating a parser would be straightforward if COMPOS were to be reimplemented in other frameworks.

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.

5 Research Approach

The research in this thesis is exploratory using *techniques*, *tools* and *scenarios*. In our context, techniques are "ideas" of how to simplify the development of IoT systems, whereas tools are the software artefacts that realise techniques. By creating

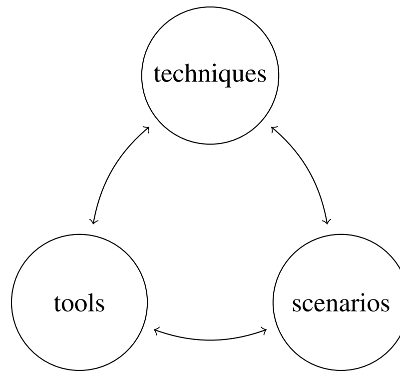


Figure 9: Iterative research approach where techniques, tools and scenarios influence each other.

a tool, we prove that it is feasible to implement a technique. Scenarios are example situations where we can try the tool and see the techniques in practice.

We work iteratively to improve our techniques, tools and scenarios. Insight from creating the tool can help us improve the techniques, and using the tool in a scenario can help us improve both the techniques and the tool but also the scenario itself. Techniques, tools and scenarios influence each other, as shown in Figure 9.

Our process for designing COMPOS began with a real-world scenario from two members of our research group that use PALCOM commercially. This scenario highlighted some problems with the current tools, and we discussed different techniques for solving them. Next, we decided what techniques we should try to implement. We then tried our tool in different scenarios. After that, we continued to refine both the tool, the techniques and scenarios iteratively.

To evaluate the techniques and tools, we sometimes use case studies. In a case study, we use our tool to implement a solution for real-world use-cases. Ideally, the real-world use-case should already have an existing solution for comparison. These kinds of case studies give knowledge about how our proposed techniques work in the real world. One drawback of case studies is the limited generalizability due to considering only one instance for the types of problems we want to solve [Cob+18].

6 Contributions

This section describes the contributions of the individual papers and the artefacts developed. But first, we describe a scenario used to relate the contributions to a concrete system.

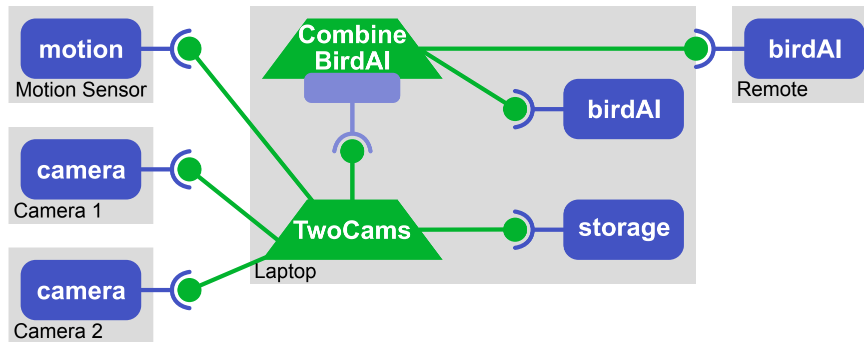


Figure 10: The figure shows an overview of a system for automatically photographing birds in a garden.

In the scenario, we build a system for taking photos of birds in a garden. The system with all the services and devices is shown in Figure 10. The idea is when the motion sensor triggers, each camera takes a photo of the garden. If one of the photos contains a bird according to one of the `birdAI` services, it is stored. The system has two compositions, `TwoCams` and `CombineBirdAI`. `CombineBirdAI` provides a synthesised service for determining if an image contains a bird or not. It does this by combining the results from the two bird detection services, one local and one in the cloud. `TwoCams` ensures that when the motion sensor triggers, the cameras take photos. If these photos contain birds, according to `CombineBirdAI`, they are stored.

Because ComPOS has evolved during this thesis, the syntax for it differs between papers. Also, compositions are called *assemblies* in papers I and IV.

Paper I: Live Programming of Internet of Things in PALCOM

When we start building the bird watcher system, the cameras and motion sensors are already used in a burglary detection system, meaning that we already have them running. To program the system against these running devices, we propose using live programming. Live programming allows the programmer to evolve running systems. We discuss using live programming to compose IoT systems in PALCOM in Paper I.

We divided the PALCOM live programming experience into three phases: explore, assemble and expose. The explore phase is about discovering and interacting with services to explore their functionality. For example, in this phase, we interact with Camera 1 to understand its interface. The assemble phase is about composing services using our DSL, for example, connecting the motion sensor to the cameras. The expose phase exposes a composition's functionality as a new

service, i.e., creating a synthesized service. We are in the expose phase when we create `CombineBirdAI`'s 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 I:

- 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.

Paper II: COMPOS: Composing Systems of Services

In Paper II, we propose a new DSL for compositions called COMPOS. We use COMPOS to specify the composition in the bird watching system. In the DSL, connections to services and devices are specified, making dependencies between devices explicit, such as the dependency between the motion sensor and the laptop.

A COMPOS script contains a list of event handlers. When a service spontaneously sends out a message that matches an event, e.g. motion in the garden detected, the composition starts to execute the reaction. The reaction contains actions for sending and receiving messages, e.g. sending a request to take a photo and receiving the photo. The reaction can also express alternatives, e.g. do different things if a photo contains a bird or not. Moreover, it can do actions in parallel, such as taking multiple photos.

When a event matches, we chose, in some cases, to abort an already running reaction triggered by the same service (see the paper for the cases when we abort). We motivate why we choose the abort strategy by using weak connectivity as an argument. For cases where the system designer desires other semantics than to abort, we show how strategy services can adapt compositions' semantics.

In this paper, we build the bird watching system to illustrate the features of COMPOS. To further evaluate COMPOS, we use it in an e-health scenario not designed by us. We also discuss how we can implement the home automation scenarios identified by Rodríguez-Avila, de Koster, and de Meuter [Rdd21].

List of contributions in Paper II:

- COMPOS, a new DSL for composing IoT services with nested and parallel message sequences.
- Different extensions of a bird watching scenarios showing practical uses of COMPOS.

- Four different strategies for handling new messages when existing reactions are blocked: *abort*, *parallel*, *ignore*, and *queue*.
- An implementation of the abort strategy using *epochs* for keeping track of related reactions in different processes.
- Strategy services that can be used to implement other strategies than abort.
- A case study evaluating COMPOS in a commercial e-health scenario.
- Implementations of seven commonly occurring home automation scenarios identified by Rodríguez-Avila, de Koster, and de Meuter [Rdd21].

Paper III: Runtime Modeling and Analysis of IoT Systems

In the bird watching system, we want to ensure that the system functions with only one camera, if the other fails. One way to ensure such a thing is by analyzing the whole system. Our idea is to create a runtime model of the system and then analyze the model.

In Paper III, we present two runtime models for PALCOM systems. The first one is the conceptual model in Figure 1 in this chapter. The second model is an extension of the first model. We use the discovery protocol of PALCOM to instantiate the model with devices, services and compositions found on the network. A composition may have connections to services that PALCOM has not discovered. The extended model can represent these undiscovered devices and services.

To allow for better analysis of PALCOM IoT systems, we have added the compositions' AST to the model. We specify the model using Relational RAGs. Due to Relational RAGs being a superset of RAGs, we can reuse attributes from the JASTADD specification for COMPOS when writing our analysis.

To demonstrate analysis using the model, we specify an analysis that can ensure that the bird watching system works with only one camera. We call this analysis device dependency analysis (DDA), and it finds sets of connected devices needed for a specific event to happen. Paper III uses another example than the bird watching system, but after reading the paper, it is hopefully clear how DDA can ensure that the system works with only one camera.

List of contributions in Paper III:

- A basic runtime model for the PALCOM IoT architecture, formalized using Relational RAGs.
- A home automation scenario as a motivating example.
- An extended runtime model that can handle incomplete systems (where devices can be unavailable) and that includes composition scripts that enable more fine-grained analyses than the basic model.

- Introducing and formalizing the Device Dependency Analysis (DDA), and showing how it can be specified using Relational RAGs on top of the extended runtime model.

Paper IV: JATTE: A Tunable Tree Editor for Integrated DSLs

When developing the COMPOS language, we wanted to experiment and change the language and the editor quickly. The experimentation with the language can be seen by the various renditions of it in the different papers. To allow this, we created JATTE.

In Paper IV, 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 equations that 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 IV:

- 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 attribute equations.
- 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.

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 available 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`. Most of the code[1] for COMPOS and the PALCOM browser with COMPOS is available at `https://bitbucket.org/palcom/compos-artefact`.

7 Conclusions and Future Work

In this thesis, we explore ways to simplify the development of IoT systems. IoT systems can be hard to understand with nontrivial dependencies between devices. Our approach is to design a DSL, called COMPOS, that makes the system's connections explicit, thereby making dependencies easier to analyze. We demonstrate the analyzability of COMPOS by designing and implementing the device dependency analysis. IoT systems may use unstable networks; hence, COMPOS is designed to handle connections going up and 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 find more ways to analyze a connected system. One idea is adding more expressive interface descriptions to services for analyzing how messages flow through a system—another idea is to generate overviews similar to Figure 2. One could also analyze security aspects of a system, such as the information flow [Den76]. The second line of research is to see how COMPOS generalizes to another IoT framework, such as ZigBee. 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. User studies can 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 levels 5 and 6 by leveraging opportunistic composition engines [You+18], e.g. automatically generating the `Two Cams` composition just given the services to compose.

[1] As of the spring 2021, you can not build COMPOS from the provided code due to PALCOM not being publicly available at this time.

References

[AP02] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java*. 2nd ed. 32 Avenue of the Americas,New York: Cambridge University Press, 2002, pp. 203–204.

[AB05] Cyrille Artho and Armin Biere. “Combined Static and Dynamic Analysis”. In: *Electronic Notes in Theoretical Computer Science* 131 (2005). Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL 2005), pp. 3–14.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A survey”. In: *Computer Networks* 54.15 (2010), pp. 2787–2805.

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

[Cli+00] 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.

[Cob+18] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. “Interdisciplinary Programming Language Design”. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Boston, MA, USA: Association for Computing Machinery, 2018, 133–146.

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

[Den76] Dorothy E. Denning. “A Lattice Model of Secure Information Flow”. In: *Commun. ACM* 19.5 (May 1976), 236–243.

[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.

[Ern03] Michael D. Ernst. “Static and Dynamic Analysis: Synergy and Duality”. In: *IN WODA 2003: ICSE WORKSHOP ON DYNAMIC ANALYSIS*. 2003, pp. 24–27.

[Eug+03] 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.

[Gis08] "CHAPTER 6 - The ZigBee Cluster Library". In: *Zigbee Wireless Networking*. Ed. by Drew Gislason. Burlington: Newnes, 2008, pp. 239–271.

[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.

[HBE11] Olivier Hersent, David Boswarthick, and Omar Elloumi. "ZigBee". In: *The Internet of Things*. John Wiley & Sons, Ltd, 2011. Chap. 7, pp. 93–137.

[Hud96] Paul Hudak. "Building Domain-Specific Embedded Languages". In: *ACM Comput. Surv.* 28.4es (Dec. 1996).

[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+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.

[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.

[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.

[Mey+20] Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. "Relational reference attribute grammars: Improving continuous model validation". In: *Journal of Computer Languages* 57 (2020). 100940.

[Ngu+17] Anne H. Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan 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.

[NN+08] Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. “Declarative Intraprocedural Flow Analysis of Java Source Code”. In: *Proceedings of the Eight Workshop on Language Description, Tools and Applications (LDTA 2008)*. Electronic Notes in Theoretical Computer Science. Elsevier B.V., 2008.

[Pel03] Chris Peltz. “Web services orchestration and choreography”. In: *Computer* 36.10 (2003), pp. 46–52.

[Rdd21] Humberto Rodríguez-Avila, Joeri de Koster, and Wolfgang de Meuter. “Advanced Join Patterns for the Actor Model based on CEP Techniques”. In: *Art Sci. Eng. Program.* 5.2 (2021), p. 10.

[Sch+19] René Schöne, Johannes Mey, Boqi Ren, and Uwe Aßmann. “Bridging the Gap between Smart Home Platforms and Machine Learning using Relational Reference Attribute Grammars”. In: *Proceedings of the 14th International Workshop on Models@run.time*. Munich, Sept. 2019, pp. 533–542.

[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.

[You+18] 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.

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 *ex-*

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

posing 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.

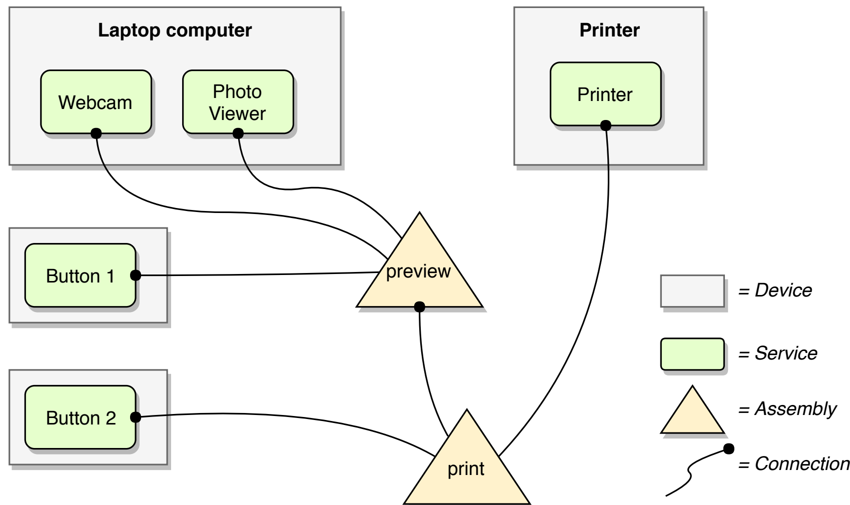


Figure 1: Overview of a PalCom photo booth application with two assemblies.

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.

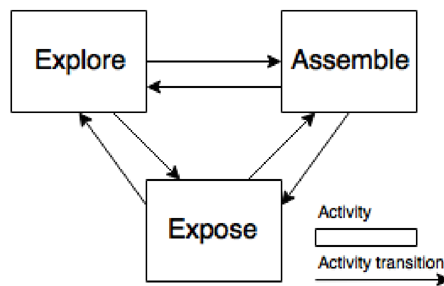


Figure 2: The activities of live programming in PALCOM

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.

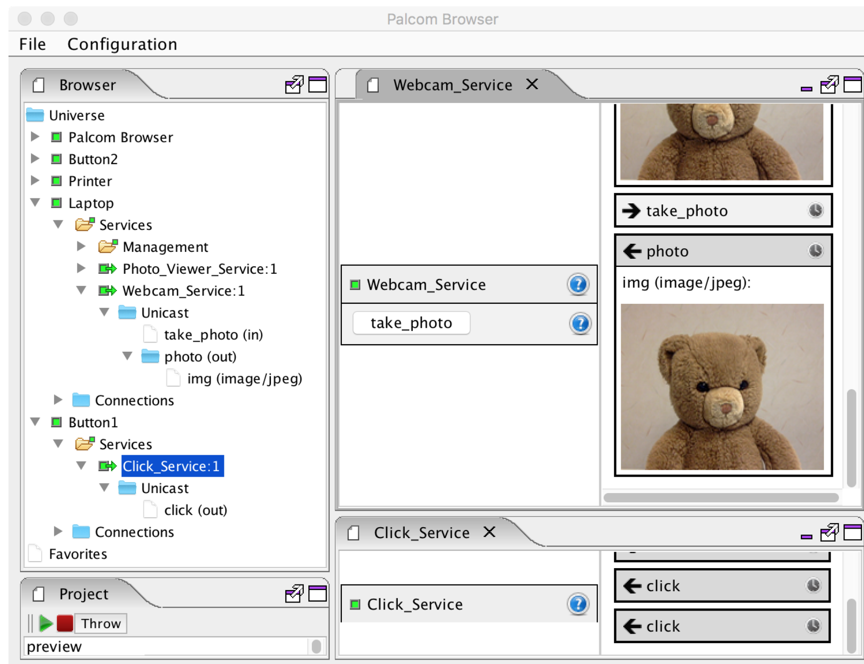


Figure 3: The PALCOM Browser. To the left, the discovery view with devices, services, and commands. To the right, two remote interaction views. One for the web camera, and one for the click service on one of the buttons.

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 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

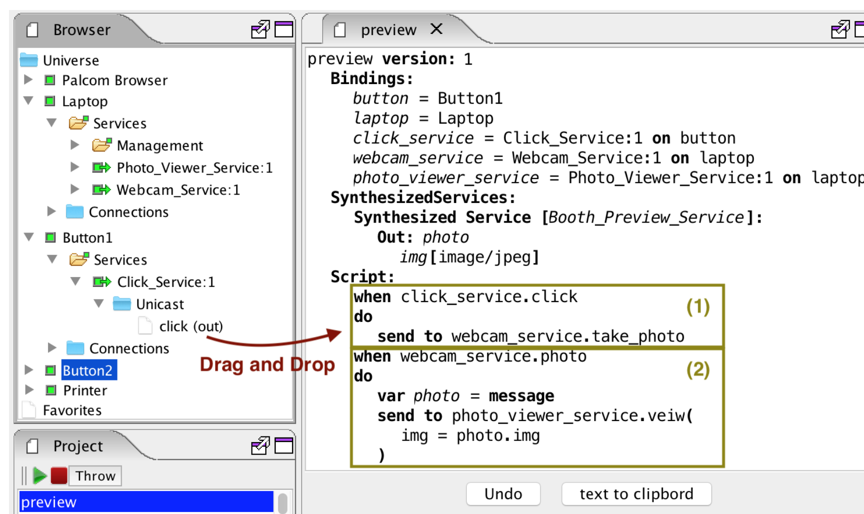


Figure 4: The PALCOM Browser in assembly code editing mode

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.

```
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

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 han-

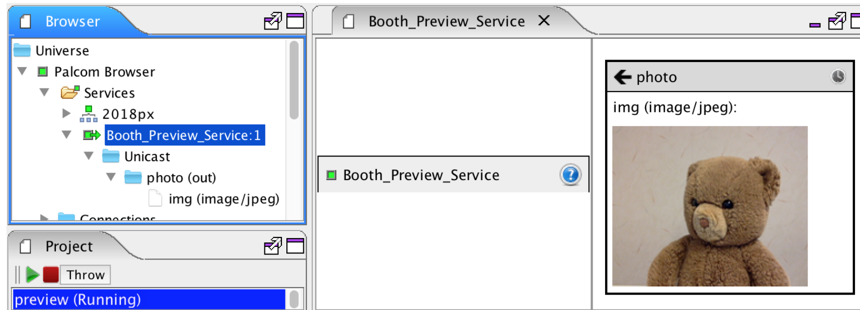


Figure 6: The remote interaction view for the synthesized service

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

```
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.

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.

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.

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 Systems of Services

Abstract

Future Internet-of-Things (IoT) systems need to be able to combine heterogeneous services and support weak connectivity. In this paper, we introduce COMPOS, a domain-specific language for composing services into IoT systems. COMPOS supports stateful reactions with nested and parallel message sequences. Our implementation of COMPOS uses abort semantics (i.e., aborting old reactions when a newer message arrives) to deal with weak connectivity. We show how to get other semantics than abort by adding strategy services. We evaluated our approach on home automation scenarios and conducted a case study based on a commercial e-health system. In the case study, we found that COMPOS makes the control flow more explicit than the composition language currently used in the system.

1 Introduction

Current IoT systems typically have a cloud-centric architecture, where sensor devices stream data to cloud servers. In these architectures, computation and storage takes place in the cloud, and user applications interact with the data in the cloud. This leads to IoT platform silos, where each system works in isolation, and where it is difficult to compose data, services, and devices from different silos into new

Extended version of Alfred Åkesson, 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

systems [Der+15; Che+14; PLM14]. Future IoT systems are expected to contain more powerful devices, with more computation taking place at the edge of the network, and with a need for handling unreliable (weak) connectivity of heterogeneous networks in a robust way [TM17].

We are exploring how to program such new kinds 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 work is based on the PALCOM IoT middleware [SF09; SF+09] which supports asynchronous message passing between processes on devices.

PalCom supports a component-connector IoT architecture with two kinds of components: *services* that perform computations and interact with the physical world, and *compositions* that connect to zero or more services, and mediate and adapt messages between these services. A service can only communicate with compositions, and not directly with other services. Services do not take the initiative to connect to other processes, and are in this sense *oblivious* to with whom they are communicating.

Services are normally implemented in an ordinary general-purpose language, e.g., Java, whereas the compositions are implemented using a domain-specific language (DSL). Two goals of using a DSL for the compositions are making it easy to *construct* new applications by composing services and making it easy to *understand* a distributed IoT system by having connections and message sequencing explicit in the DSL.

Compositions can also expose functionality as so called *synthesized* services, allowing other compositions to connect to them, forming hierarchical compositions. Ordinary services, i.e., services that are not synthesized by a composition, are called *native* services.

The original composition language used in PALCOM is a simple event-driven language [SHM07] with very limited expressibility. For example, only sequential message composition is supported. All incoming events are handled by unrelated stateless reactions.

In this paper, we propose a new composition language, COMPOS (Composition language for PALCOM Oblivious Services), that enables more expressive compositions. COMPOS supports sequential, parallel, and nested message sequences, and has support for both request-reply and solicit-response message patterns. Reactions are stateful, and may block when waiting for messages. This way, they may be related to reactions in other compositions.

While COMPOS is more powerful than the original language, we still want to keep the design goals of having a very simple language that is focused on handling coordination and mediation of messages, and where computation is delegated to native services. For this reason, COMPOS does not contain general-purpose computational constructs.

With the addition of stateful reactions, the question arises of how to handle weak connectivity, when awaited replies never arrive, or arrive too late to be meaningful for the application. Different strategies can be used for handling this problem. For COMPOS, we have chosen an *abort* semantics that removes an ongoing blocked reaction when a new related message arrives. We demonstrate how this strategy limits the simultaneously ongoing activity in the system, while giving preference to newer information. We also demonstrate how other strategies can be obtained by adding strategy services.

We start by giving some background on the component-connector IoT architecture and different message patterns used (Section 2), and a running example for an IoT system using this architecture (Section 3). We then present the main contributions of the paper:

- We introduce the COMPOS language with nested and parallel message sequences (Section 4).
- We show different practical uses of COMPOS by extending the running example (Section 5).
- We identify four different strategies for handling new messages when existing reactions are blocked: *abort*, *parallel*, *ignore*, and *queue*. We implement the abort strategy by introducing *epochs* for keeping track of related reactions in different processes (Section 6).
- We demonstrate how strategy services can be used to implement other strategies than abort (Section 7).
- We evaluate the language by applying it to real-world scenarios (Section 8). First, we used COMPOS to re-implement a scaled-down version of a commercial e-health application. Second, we discuss how we can implement seven commonly occurring home automation scenarios, as identified by Rodríguez-Avila, de Koster, and de Meuter [Rdd21].

We end the paper by presenting related work in Section 9 and conclude in Section 10.

2 IoT Architecture

In this section, we present the component-connector architecture used in PALCOM, and the message patterns that we will use for COMPOS.

2.1 Services and Compositions

Figure 1 shows an example PALCOM IoT system consisting of four devices, d_1, d_2, d_3, and d_4. Each device can host a number of native services and compositions.

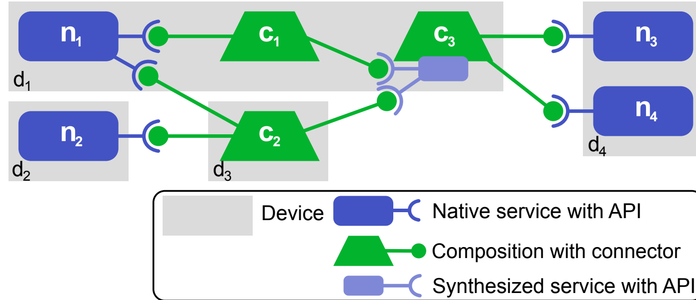


Figure 1: System of devices with services and compositions.

For example, d_1 hosts the native service n_1 and the two compositions c_1 and c_3. Native services implement computations and interactions with the physical world, whereas compositions mediate and adapt messages between services.

Each composition connects to zero or more services. For example, c_3 connects to n_3 and n_4. It is possible for more than one composition to connect to the same service, e.g., both c_1 and c_2 connect to service n_1.

Each service has an API of signatures for the incoming messages it can receive and the outgoing messages it can send. This information is discovered by the middleware, and can be used when designing a composition.

A composition may have a *synthesized* service to which other compositions can connect. This allows reusable compositions to be built, and composed in a hierarchy. For example, instead of c_1 and c_2 each handling the mediation to n_3 and n_4, this mediation is encapsulated in c_3, and can be reused via the synthesized service of c_3.

2.2 Port and Cable Metaphor

Metaphorically, we can think of native services as ports on physical devices, and compositions as multiway adaptor cables that connect these ports. A composition with a synthesized service corresponds to an adaptor cable that has a dongle on the middle, with a port into which other cables can plug in.

Metaphors from the physical world often need to be enhanced with some degree of “magic” to better fit a computational system [Smi87]. An example of this is that the “cable” (composition) knows itself what “ports“ (services) it should connect to, and automatically connects itself to services within reach (via some network). This is handled by the underlying middleware.

2.3 Deployment and Configuration

Services and compositions are runtime processes. To connect to a service, a composition uses a logical address consisting of a *concrete type* of the service, a *URI* for the device where the service runs, and an *instance name*, which is a device-local name for the running service. Given this logical address, the middleware can automatically construct the physical connection, which might go via multiple physical devices and network tunnels, and using different transport protocols like TCP or UDP. The physical connection can also be transparently updated during operation, for example for a mobile phone that alternates between using Wi-Fi or cellular networks, depending on location.

The concrete type of a service is a globally unique name for the service implementation, for example a UUID [LMS05]. The URI for a device is a globally unique name for a device, constructed, for example, from the MAC address from one of its network interfaces. The instance name is local to a device, and constructed so that all instances of the same concrete service type have distinct names. The instance names are persistent, so that once deployed, a service keeps its instance name even if it is stopped and started again, or if the device is rebooted.

Services and compositions can be deployed manually by a user, or via a configuration script on a configuration server [Nor+20]. To make compositions reusable, they can be parameterized by what devices and instance names to use for their service connections. This can be defined in the configurations.

In practice, a user will typically create a new configuration manually, to test it, and then generate a script for it in order to be able to reuse it on other devices and for other service instances. Parameterization of compositions and scripting of configurations is orthogonal to the composition language, and therefore not further treated in this paper.

2.4 Acyclic Graph and Versioning

It can be noted that the graph of PALCOM compositions and services is acyclic by construction. This is because in order to define a connection in a composition, the concrete type of the service must already exist, since it is part of the logical address of the service. A composition type will thus always be newer than the types of services it connects to, and a cyclic graph cannot be formed.

This also has the consequence that in order to update a system with a new version of a service, all compositions connecting to it need to be updated as well, in order to use the new service type. However, such updates are possible to automate.

2.5 Message Patterns and Reactions

In WSDL [Chr+01], messages between a client and a server are categorized into four patterns. Here, we define the following four analogous patterns, where the

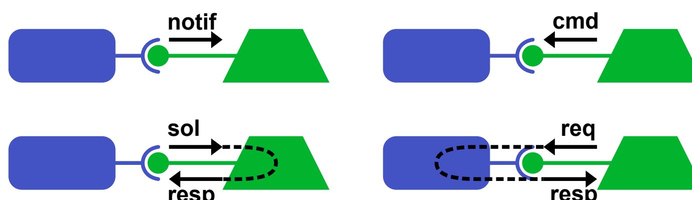


Figure 2: Message patterns with notification, command, solicit/response, and request/response.

compositions play the role of clients, and services play the role of servers. See also Figure 2.

notification (notif) A service sends a notification to all its connected compositions.

command (cmd) A composition sends a command to a specific service.

solicit/response (sol/resp) A service sends a solicit and expects to receive a response. A composition receiving the solicit will send a response back to the service. For simplicity we assume that at most one composition is connected to the service if solicit/response is used.

request/response (req/resp) A composition sends a request to a specific service, and the service will then send a response back to the composition.

A composition is purely reactive: it only takes action when it receives an incoming message—it does not take any spontaneous initiative in sending messages. All activity in a system therefore starts with a native service spontaneously sending a message to a composition: either a notification or a solicit. This starts a *reaction* in the composition that may send other messages, spreading activity throughout the system.

The first composition language used in PALCOM, called the *assembly language* [SHM07], did not distinguish between different message patterns, and can be viewed as only supporting notifications and commands. Each incoming message to a composition started a new stateless reaction that could send other messages, but not wait for responses to arrive. Simple support for request-response was added later, and included in the 4.0.19 release of PALCOM[1]. Reactions were, however, still stateless, so an incoming response was treated as starting a new stateless reaction.

The composition language introduced in this paper, COMPOS, supports all four message patterns, and has stateful reactions that can wait for responses and store variables local to a reaction.

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

2.6 Spontaneous and Expected Messages

In COMPOS, we refer to notifications, solicits, commands, and requests as *spontaneous* messages. When a spontaneous message is received, this starts a new independent reaction in the receiving composition. i.e., a spontaneous message is 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.

3 Motivating Example

As a motivating example, we will use a variant of a bird watching scenario [Åke+18a], and discuss why a stateless composition language is not sufficient.

3.1 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 later. 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.

3.2 Devices and Services Used

Figure 3 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 Wi-Fi network and they can discover each other using the PALCOM middleware.[2]

Using the discovery mechanism, Maria identifies the services on the network. The services in Maria's system are:

- A motion service that sends a notification, `move`, each time a movement is detected
- A camera service that can take a photo on request (`take_photo`) and respond with the image (`photo(img)`).

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

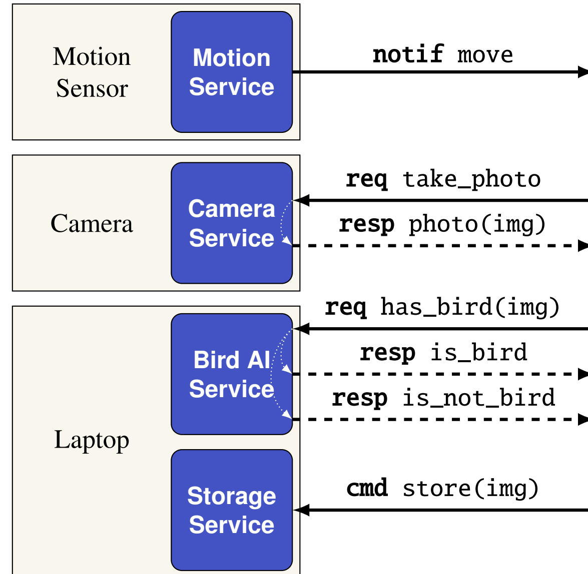


Figure 3: Services available to Maria, including APIs.

- A bird AI service that can classify an image as containing a bird or not, by receiving a request with an image (`has_bird(img)`) and responding with either the message `is_bird` or `is_not_bird`.
- A storage service that can receive a command with an image (`store(img)`) for permanent storage.

3.3 Constructing the Application

To construct the bird watcher application from the services above, Maria creates a *composition* (a COMPOS program), that connects to the relevant services, and that includes a script for how messages should be mediated.

The composition can be created in an easy way using structure editing and drag-and-drop from a service discovery browser [Åke+18b; ÅH17] called the PALCOM Browser. For instance, to write an action *send msg to service*, Maria locates the message in the browser, and drags and drops it to the script.

Maria writes the script such that when a `move` notification arrives from the motion sensor, a `take_photo` request is sent to the camera, which responds with a `photo` message. The composition then sends the request `has_bird` to the bird recognition service which responds with either an `is_bird` or an `is_not_bird`

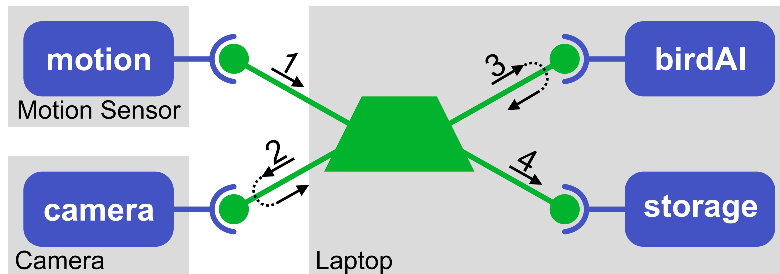


Figure 4: Overview of Maria's bird watcher system. 1) `move` notification, 2) `take_photo` request with `photo(img)` response, 3) `has_bird` request with `is_bird` or `is_not_bird` response, 4) `store(img)` command.

reply. In the case of an `is_bird` reply, the composition sends the command `store` to the storage service. Figure 4 shows an overview of the resulting system.

Maria deploys the composition 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.

3.4 The Need for Stateful Reactions

It would be difficult to write this script with stateless reactions. First, the image needs to be cached by the composition, unless we require the bird service to return the image in its reply, which would be needlessly inefficient. Second, even if the image is cached globally in the composition, we need to make sure that the reply from the bird service is actually about the cached image. This can be difficult to ensure if multiple photos are taken before the bird service has time to reply. If we construct a slightly more complex scenario with several cameras or several alternative bird services, the composition may need to wait for several responses in a reaction. In the next section, we present how the COMPOS language handles these issues.

4 The COMPOS Language

A COMPOS program contains connection declarations, declarations of synthesized services, and a coordination script for handling incoming messages. To execute COMPOS programs, 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 connection 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.

4.1 Connections

The connection part defines what running services the composition should connect to, and gives them local names to use in the coordination script, as in the following example.

```
1 composition Example
2   device laptop = "Maria's Laptop"
3   service birdAI = "Bird AI Service" on laptop
4   service storage = "Storage Service" on laptop
5   ...
```

Here, `laptop` is the local name of a specific device. The URI of this device is a long string that is omitted from view in the COMPOS editor. The string `"Maria's Laptop"` is a human-readable name of the device, used just for documentation purposes—it does not need to be unique. Further, `birdAI` and `storage` are local names of two running services on the `laptop` device. Again, the URIs for the concrete types of these services are omitted from view, and `"Bird Service"` and `"Storage Service"` are just human readable names for these types. Optionally, a local instance name can be added, in case a device runs more than one instance of the same concrete service type.

In deploying the composition, the underlying middleware automatically establishes and takes down connections, as services and devices are discovered or undiscovered. The coordination script can send and receive messages on these connections. In case a connection is down, any messages sent over it will simply be lost.

It is possible to reuse the composition in another setting, binding to other devices than those used in the script. This can be done by overriding the URIs of devices in configurations, as discussed in Section 2.3. A debugger may then show the human-readable names of those overriding devices instead of the ones defined in the composition.

4.2 Synthesized Services

A synthesized service defines an API to which other compositions can connect, as in the following example.

```
1 composition Example
2   ...
3   synthesized service Example
4     in ma (arg1:Type1, arg2:Type2)
5     out mb ()
6     out mc ()
7     pattern ma -> (mb | mc)
8     ...
```

A synthesized service can be given the same name as the composition, as in the example above. But if there is more than one synthesized service, they must have distinct names.

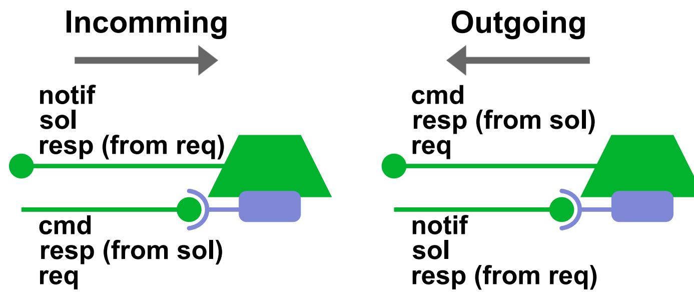


Figure 5: Incoming and outgoing messages to/from a composition.

The API defines *in* messages that can be sent to the synthesized service, and *out* messages that the composition can send from the service, to the compositions that connect to it. Each message can have zero or more typed arguments. The `pattern` clause defines the message patterns used. For example, in this case, `ma` is a request that replies with either `mb` or `mc`. For notifications and commands, no pattern clauses are needed. Patterns are currently not supported by PALCOM and not enforced in COMPOS.

4.3 Message Kinds

Because of the different message patterns with in total six kinds of messages, and the fact that a composition can both have its own connections and be connected to via its synthesized services, there are 12 different kinds of possible messages, as shown in Figure 5.

In COMPOS scripts, messages in `send` and `receive` constructs are tagged with message kinds, i.e., `cmd`, `req`, `resp`, `sol`, or `notif`. These tags are not strictly necessary, as they could be inferred from the APIs, and are only included for clarity.

4.4 Coordination Script

A COMPOS coordination script consists of a set of event handlers for incoming messages, so called *when-dos*. Each *when* part matches an input message either *from* a specific service or *to* a synthesized service of the composition, as in the following example.

```
1  composition ...
2    service s = ...
3    synthesized service T ...
4
5    when notif n from s do
6        ...
7
8    when cmd c to T do
9        ...
```

All *when* parts must be mutually exclusive, so when a message arrives, there is only one event handler 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*. The following actions are supported:

send Non-blocking. Sends a message to a receiver.

receive Blocking. Waits for a response from a previous request or solicit.

select Contains a set of mutually exclusive event handlers for incoming messages, like the `when` at the top level of the script. However, in a *select* the inputs are responses, whereas at the top level they are spontaneous messages. Blocks until one of the event-handlers matches.

parallel Runs a set of action sequences in parallel. Blocks until all its sequences are finished.

finish first Similar to the *parallel* action, but blocks only until one of its sequences has finished.

The following example illustrates *send*, *receive* and *select*.

```
1    service s = ...
2    when ... do
3      send req r1 to s
4      receive resp r1answ(var p) from s
5      send req r2(p) to s
6      select
7        when resp r2answ1 from s do
8          ...
9        when resp r2answ2 from s do
10         ...
11     end
```

Here, the *do* part starts by sending the request `r1` to `s`. Then the reaction blocks until the response `r1answ` is received. The `r1answ` includes a parameter `p` that is captured in the script using the `var` construct. This parameter is then used when sending a new request `r2` to `s`. The reaction then blocks on the *select* action, until one of the responses `r2answ1` or `r2answ2` arrives.

It can be noted that request-responses are not syntactically coupled like remote procedure calls would be. However, for each response, the corresponding send can be statically determined from the script (the nearest previous send to the same service). If a corresponding send is missing, there is a static error in the script, which is flagged by the COMPOS editor.

It can also be noted that COMPOS does not include any constructs for general-purpose computations. Such computations would need to be delegated to a service. Conditionals are supported through the select action, by reacting differently depending on which response message is received.

The listing below shows an example of the `parallel` construct. The parallel branches are separated with the keyword `and`. The enclosing action sequence continues to execute when all branches have finished.

```
1   ...
2   parallel
3     send req r1 ...
4     receive resp r1answ ...
5     ...
6   and
7     send req r2 ...
8     receive resp r2answ ...
9     ...
10  end
11  ...
```

The following listing shows the `finish first` construct. Here, the branches are separated by `or`, and the enclosing action sequence continues to execute when one of the branches has finished.

```
1   ...
2   finish first
3     send req r1 ...
4     receive resp r1answ ...
5     ...
6   or
7     send req r2 ...
8     receive resp r2answ ...
9     ...
10  end
11  ...
```

Practical use of the different constructs will be illustrated in Section 5. A specification of the COMPOS syntax in EBNF can be found in appendix A.1.

4.5 Reactions

The COMPOS interpreter has a queue for incoming messages, and handles the messages in order of arrival. For each message, the interpreter will either create a new reaction, continue the execution of a previously blocked reaction, or simply ignore the message (when there is no suitable action to take).

When receiving a spontaneous message that matches a when-do clause at the outermost level, the interpreter creates a new *reaction* for the corresponding reaction specification. The reaction contains local variable values and a program counter, keeping track of the currently executing action. The reaction may also contain subreactions with their own program counter and local variables, see below.

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.[3]

For *parallel* and *finish first* actions, one subreaction is created for each parallel action subsequence, with its own program counter and local variables. The interpreter executes each of the subreactions until it blocks, and after that, the parent reaction also blocks. Later, when a new message arrives that one of the subreactions is waiting for, that subreaction continues to execute until it finishes or blocks. In case the subreaction finishes, and was the last/first subreaction to finish in a *parallel/finish-first* action, the parent reaction will continue its execution.

We can note that the *root* reaction in a subreaction hierarchy is started by a when-do at the outermost level in the script.

4.6 Resuming a Reaction

A reaction is resumed when a response message it is waiting for is handled. To be able to identify the reaction to resume, each outgoing request/solicit message is tagged with a *sending-reaction-id*, identifying the root reaction of the reaction sending the message, and a *send-id*, identifying the send action used (a position in the script). These values are returned by the response message.

Each reaction that is waiting for a response message is blocked at either a receive or a select action. From this receive or a select action it is straightforward to compute the corresponding send-id.

To handle a response, the root reaction is first identified using the sending-reaction-id in the response message. Then, any blocked action under this root reaction that matches the name and send-id of the response message continues to execute. It can be noted that there can be at most one match because two subreactions are not allowed to wait for the same response messages.

If a matching reaction is found, it will be resumed in order to consume the response message. If no matching reaction is found, the response message is ignored. This can happen, for example, if the composition has been restarted since

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

the request/solicit was sent, or if a waiting subreaction has already terminated due to an outer finish-first action.

Pseudocode for the interpreter can be found in appendix A.2.

5 Composing Scenarios

In this section, we will illustrate practical use of COMPOS by discussing different variants of the birdwatcher scenario: the simple scenario from Section 3, a scenario that uses the parallel construct to handle two cameras, a refactored scenario adding a composition with a synthesized service to avoid duplication of code, and a scenario illustrating nested `parallel` and `finish first` actions.

5.1 Simple Birdwatcher

The listing below shows the script for the simple birdwatcher composition from Section 3. The logic is quite simple: when a `move` message arrives from the `sensor`, a request is sent to the `camera` to take a photo. When the response arrives, the image is sent to the `birdAI`. Depending on the response, the image is stored or not. Then the reaction terminates.

```
1  composition SimpleBirdwatcher
2    service sensor = ...
3    service camera = ...
4    service birdAI = ...
5    service storage = ...
6    when notif move from sensor do
7      send req take_photo to camera
8      receive resp photo(var img) from camera
9      send req has_bird(img) to birdAI
10     select
11       when resp is_bird from birdAI do
12         send cmd store_image(img) to storage
13       when resp is_not_bird from birdAI do
14     end
```

Figure 6 shows a sequence diagram for the composition, illustrating when the reaction is blocked (waiting for a response).

In Section 6.1 we will discuss what happens if a new `move` arrives before an ongoing reaction has terminated.

5.2 Two Cameras in Parallel

To illustrate the *parallel* construct, we suppose that Maria adds one more camera, to see the birds from more angles. She replaces[4] the `SimpleBirdwatcher` com-

[4]In practice, Maria would typically edit the birdwatcher composition rather than replacing it. For clarity, we use different names for all versions of the compositions.

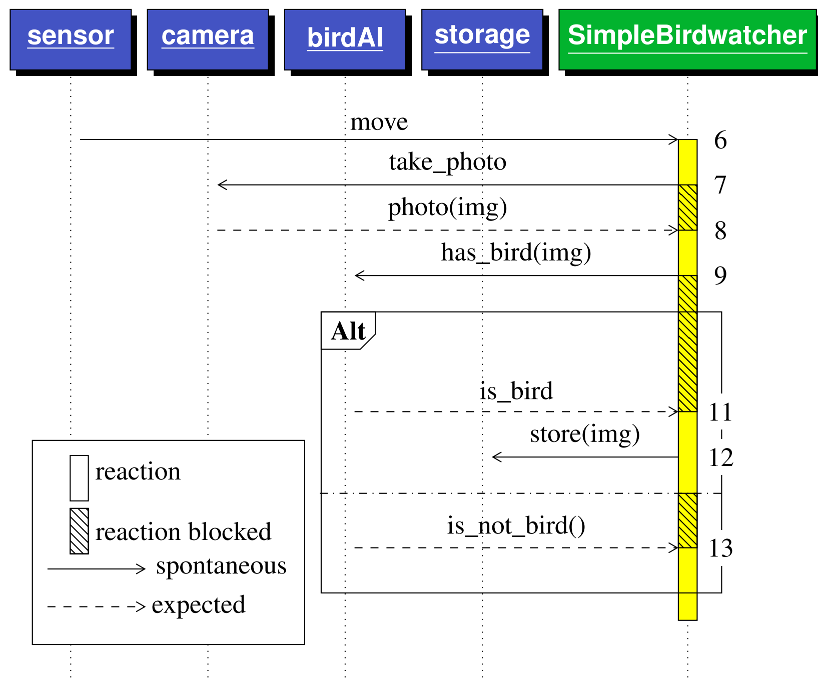


Figure 6: Sequence diagram for `SimpleBirdwatcher`. The number to the right is the corresponding line number in `SimpleBirdwatcher`.

position with a new composition `TwoCamBirdwatcher` which uses the `parallel` action to take and process pictures in parallel:

```
1  composition TwoCamBirdwatcher
2    // declare sensor, camera_1, camera_2, birdAI, storage
3
4    when notif move from sensor do
5      parallel
6        send req take_photo to camera_1
7        receive resp photo(var img) from camera_1
8        send req has_bird(img) to birdAI
9        select
10         when resp is_bird from birdAI do
11           send cmd store_image(img) to storage
12         when resp is_not_bird from birdAI do
13       end
14     and
15       send req take_photo to camera_2
16       receive resp photo(var img) from camera_2
17       send req has_bird(img) to birdAI
18       select
19         when resp is_bird from birdAI do
20           send cmd store_image(img) to storage
21         when resp is_not_bird from birdAI do
22       end
23     end
```

5.3 Factor out Duplicated Code from a Composition

The `TwoCamBirdwatcher` composition has duplicated code that Maria would like to avoid: the code interacting with `birdAI` and `storage` (lines 8-12 and 17-21) is the same for both parallel branches.

It is possible to use a composition with a synthesized service as an abstraction mechanism to solve this problem. Common code can be factored out to a separate composition, and the combined functionality can be provided to the rest of the system by a synthesized service.

Maria uses this approach to factor out the duplicated code into a new composition `StoreIfBird` with a synthesized service. The refactored version of the original composition, `TwoCamRefactored`, interacts with `StoreIfBird` instead of directly with `birdAI` and `storage`. See Figure 7.

The refactored code is shown in the following listings. Note that a reaction in the `StoreIfBird` composition starts by an incoming command message on its synthesized service. Since `TwoCamRefactored` sends two such commands in parallel, this gives rise to two independent reactions in `StoreIfBird`.

```
1  composition TwoCamRefactored
2    // declare sensor, camera_1, camera_2, store_if_bird
3
4    when notif move from sensor do
5      parallel
6        send req take_photo to camera_1
7        receive resp photo(var img) from camera_1
8        send cmd an_image(img) to store_if_bird
9      and
10       send req take_photo to camera_2
11       receive resp photo(var img) from camera_2
12       send cmd an_image(img) to store_if_bird
13     end
```

```
1  composition StoreIfBird
2    // declare birdAI, storage
3
4    synthesized service StoreIfBird
5      in an_image(img:jpg)
6
7    when cmd an_image(var img) to StoreIfBird do
8      send req has_bird(img) to birdAI
9      select
10       when resp is_bird from birdAI do
11         send cmd store_image(img) to storage
12       when resp is_not_bird from birdAI do
13     end
```

5.4 Nested Actions

We will now illustrate nested *parallel* and *finish first* actions, and show how the decoupling of sends and receives can be utilized.

Maria finds out that her `birdAI` service gives too many false negatives, and she wants to combine it with a better (but usually slower) one that she finds online. She creates a new composition, `CombinedAI` that has a synthesized service with the same interface as her local `birdAI` service. The new composition consults both the local and the online services. If one of them responds positively, the synthesized service replies with `is_bird`, and if both are negative, it replies with `is_not_bird`. The listing is shown below:

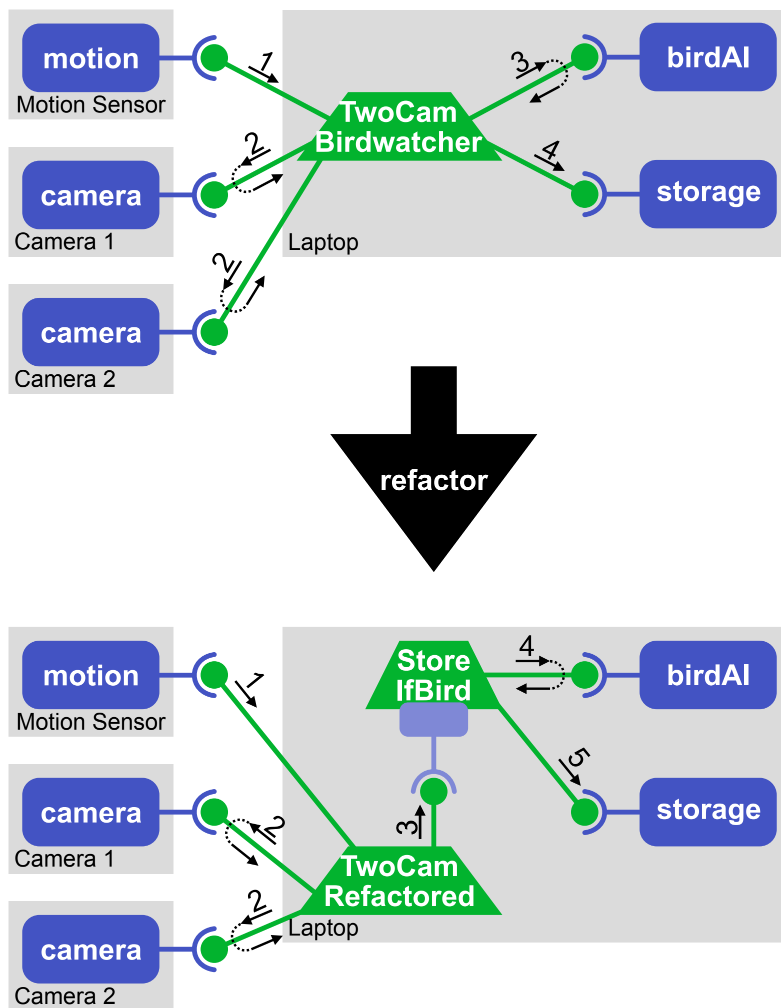


Figure 7: The two-camera system before and after refactoring. The labels on the connections indicate the order of the interaction and the direction of the messages.

```
1  composition CombinedAI
2    service local_AI = ...
3    service remote_AI = ...
4
5    synthesized service CombinedAI
6      in has_bird(img:jpg)
7      out is_bird()
8      out is_not_bird()
9      pattern has_bird -> (is_bird | is_not_bird)
10
11   when req has_bird(var img) to CombinedAI do
12     send req has_bird(img) to local_AI
13     send req has_bird(img) to remote_AI
14     finish first
15       receive resp is_bird from local_AI
16       send resp is_bird from CombinedAI
17     or
18       receive resp is_bird from remote_AI
19       send resp is_bird from CombinedAI
20     or
21       parallel
22         receive resp is_not_bird from local_AI
23       and
24         receive resp is_not_bird from remote_AI
25       end
26       send resp is_not_bird from CombinedAI
27     end
```

We can note that by decoupling the sends and receives in the script, it is possible to wait for different responses at different locations in the script. This way, the composition can respond as soon as it gets a positive result from one of the AI services.

The `CombinedAI` can now be used in one of the birdwatcher compositions, e.g., `SimpleBirdwatcher`, simply by replacing the binding of the `birdAI` service to the synthesized service of `CombinedAI`.

6 The Abort Strategy

6.1 Handling Overload and Weak Connectivity

Because of weak connectivity, messages may be delayed or lost. Even if reliable connections are used, it may happen that a remote service does not reply as expected, due to bugs or crashes. If no special measures are taken, a reaction waiting for such a reply would be *blocked forever*. A related problem is *message overload*, when a device does not have the capacity to handle all incoming messages. Furthermore, if the capacity is limited, there is the question of what *priority* to give to different reactions: should existing reactions be allowed to continue to execute,

even if we do not know if they will finish, or should they be aborted, risking valuable work to be lost? For COMPOS, a fourth issue to consider is that reactions in different compositions may be *related*, if one reaction is the result of a message sent from another composition.

There are many possible strategies for dealing with these problems. Below, we list four basic strategies for dealing with incoming spontaneous messages on a connection, in the case there is already an ongoing reaction associated with the connection.

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 for COMPOS, we have chosen the *abort* strategy. This way, blocked reactions are automatically removed when the next message arrives. Strategies *parallel*, *queue*, and *ignore* would need to be complemented with timeouts to remove blocked reactions, and such timeouts may be difficult to tune. The *abort* strategy furthermore automatically gives a bound on the simultaneously ongoing activity in the system. So does *queue* and *ignore*, but they prioritize older messages, whereas *abort* prioritizes newer ones.

For *abort* to work for hierarchical compositions (connected via synthesized services) it is important that dependent reactions are not aborted unless the original reaction is aborted. Otherwise, the parallel and finish-first constructs would not work correctly.

In some situations, one of the other strategies, *parallel*, *queue*, or *ignore*, would be more suitable. For example, if spontaneous messages arrive very frequently, *abort* might lead to ongoing reactions never getting time to complete, and *ignore* would be a better option. It turns out that by choosing *abort* as the default strategy, the other strategies can be implemented using additional strategy services. This will be discussed in Section 7. Table 1 summarizes the properties of the different strategies.

6.2 Rules for Aborting Reactions

A notification or solicit from a native service can lead to a number of reactions in several compositions, similar to a call graph. The abort semantics applies to a complete such call graph, so that a newer related call graph aborts a previous one, but reactions within the same call graph never abort each other, and reactions on unrelated call graphs should not abort each other either. We will now motivate and

	parallel	*queue*	*ignore*	*abort*
Responsive	Yes	No	No	Yes
Bounded	No	No	Yes	Yes
Eager abort	No	No	No	Yes
Need timer	Yes	Yes	Yes	No

Table 1: Summary of the properties for the different strategies. *Responsive* means that the latest message is always handled directly in the composition. *Bounded* means bounded memory use without using time-outs in cases where native services send an unlimited number of spontaneous messages. *Eager abort* means that reactions that could have finished can get removed without the use of time-outs. *Need timer* means that the strategy needs to use time-outs to remove reactions that are stuck.

illustrate this semantics by a number of smaller scenarios, formulating observations for when reactions are aborted or not, and finally formulating a general rule for the abort semantics.

6.2.1 Notifications and Solicits on the Same Connection

We begin by considering what happens when a composition connects to a service, and more than one notification/solicit arrive on the same connection. For illustration, we use a home automation scenario taken from Rodríguez-Avila, de Koster, and de Meuter [Rdd21]:

> *Send a [alert] notification when a window has been open for over an hour.*

In our solution to this scenario, we assume there are the following services:

`window` sends a notification when the window opens or closes

`timer` which can be sent a request to keep the time, and responds when the time is up

`alert` which can be sent a command to alert the user

We can implement the scenario by gluing together the services using a composition, `AlertWindowOpen`, making use of the abort semantics:

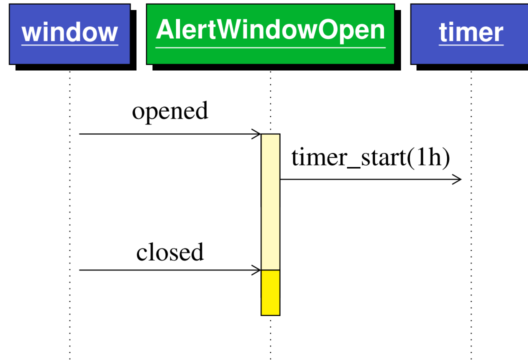


Figure 8: The `closed` message starts a new reaction, aborting the reaction started by the `opened` message.

```
1  composition AlertWindowOpen
2    service window = ...
3    service timer = ...
4    service alert = ...
5    when notif opened from window do
6     send req timer_start("1h") to timer
7     receive resp timer_end from timer
8     send cmd alert("Window open 1h") to alert
9    when notif closed from window do
```

Suppose that an `opened` notification arrives and starts a reaction in the composition. The reaction will block, waiting for the `timer` to respond (line 7). Suppose a new notification (`opened` or `closed`) arrives from `window` while the reaction is blocked. The reaction will then be aborted and a new reaction started, sending a new request to the timer. In the case of `closed`, the new reaction immediately terminates, and no alert will be sent. The sequence diagram in Figure 8 depicts this situation. However, if the window is not closed, the `timer` will respond after one hour, and the reaction will send the alert.

The abort semantics is based on that we view notifications (or solicits) that arrive on the same connection to be related, and that they can be seen as reporting an updated newer status (or sending an updated solicit). The newest message received therefore aborts a reaction started by an earlier message. We can make the following observation:

Observation 1 (Abort when newer notification has same connection)
A notification or solicit will abort a reaction started by a notification or solicit on the same connection.

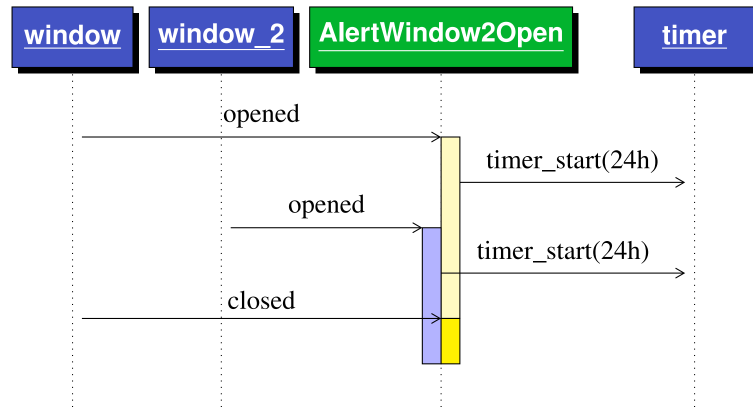


Figure 9: The two reactions run simultaneously because they are started by `opened` messages arriving on different connections.

6.2.2 Notifications and Solicits on Different Connections

How do we handle notifications and solicits coming from different services? Because these messages are unrelated, it seemed sensible for us that they should not abort each other. This gives the following observation:

Observation 2 (No abort when notification has different connection) *Notifications and solicits arriving on different connections should not abort each other's reactions.*

To illustrate this, we could add another window to our composition, `window_2`, by duplicating the behavior for that window as follows:

```
1  // Handle original window:
2  when notif opened from window do
3   send req timer_start("1h") to timer
4   receive resp timer_end from timer
5   send cmd alert("Window open 1h") to alert
6  when notif closed from window do
7  // Handle window_2:
8  when notif opened from window_2 do
9   send req timer_start("1h") to timer
10  receive resp timer_end from timer
11  send cmd alert("Window 2 open 1h") to alert
12 when notif closed from window_2 do
```

The sequence diagram in Figure 9 shows how an `opened` message from each window results in simultaneous reactions in the composition, and how a `closed` message aborts one of these reactions.

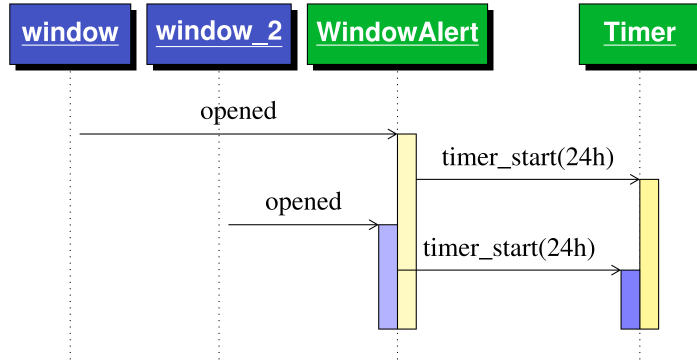


Figure 10: The two requests sent to `Timer` have different origins. Therefore, the second request starts a new reaction in `Timer`, without aborting the existing reaction.

6.2.3 Commands and Requests with Different Origin

We will now look at compositions with synthesized services. Incoming requests and commands to the synthesized service will start reactions in the composition. When should they abort each other?

Let us consider a composition `WindowAlert` which connects to `window` and `window_2` as in the previous example, but which uses a synthesized service `Timer` instead of the native `timer` service. The two requests to `Timer` are sent on the same connection, but they are unrelated and should not abort each other's reactions in the `Timer` composition. See Figure 10.

In general, when we have several compositions connected through synthesized services, some reactions will be related to each other and some will be unrelated.

To talk about unrelated requests, we introduce the notions of *source service*, *source connection*, and *source reaction*. A notification or solicit sent from a native service can start reactions in a composition. These reactions can lead to other reactions by sending commands or requests to synthesized services of other compositions. Similarly, these new reactions can lead to more new reactions, and so on. This creates a hierarchy of compositions sending commands and requests to each other, similar to a call graph. In this hierarchy, every reaction can be traced back to a native service sending a notification or solicit; we call this the *source service*. In Figure 10, `window` and `window_2` are examples of source services. The connection that a composition sets up to a source service is called a *source connection*. The reaction the composition creates when receiving a notification or solicit on the source connection is called a *source reaction*. Any reaction in a composition can be traced back to a source connection which we call the *origin* of the reaction.

Using this new terminology, we have the following observation:

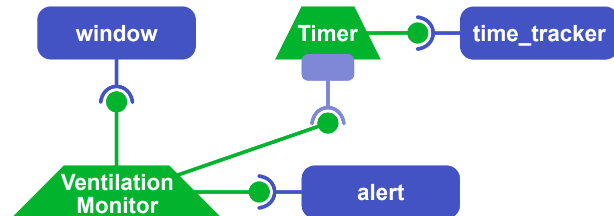


Figure 11: The figure shows an overview of window monitoring system. Devices are omitted.

Observation 3 (No abort when command has different source connection)
Commands and requests originating from different source connections should not abort each other's reactions.

6.2.4 Commands and Requests With the Same Origin

To illustrate another situation of interest, we describe a more complex scenario involving the same services. In this scenario, we want to build a system that reminds us to open and close a ventilation window. We must open the window for five to ten minutes every day for fresh air. An overview of the system is shown in Figure 11.

The `VentilationMonitor` composition composes this system and relies on the abort semantics. When the window opens, the `VentilationMonitor` starts two timers in parallel for five and ten minutes, respectively. After five minutes, the system alerts that it is time to close the window, and sends a reminder after ten minutes. When the window closes, `VentilationMonitor` sets the timer for 24h and then alerts when it is time to open the window.

```
1  composition VentilationMonitor
2  // Setup omitted
3    when notif opened from window do
4      parallel
5        send req timer_start("5min") to timer
6        receive resp timer_end from timer
7        send cmd msg("Please close the window.") to alert
8      and
9        send req timer_start("10min") to timer
10       receive resp timer_end from timer
11       send cmd msg("Please close the window now!") to alert
12     end
13   when notif closed from window do
14     send req timer_start("24h") to timer
15     receive resp timer_end from timer
16     send cmd msg("Please open the window.") to alert
```

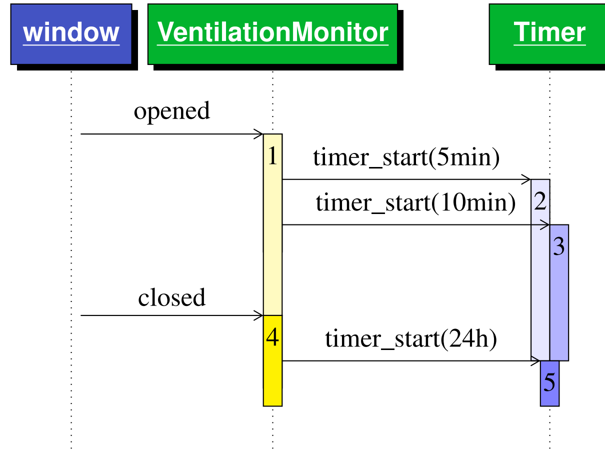


Figure 12: The `closed` message aborts reaction 1, and `timer_start(24)` aborts both reactions 2 and 3.

In this scenario, we send two requests to `Timer` in parallel (lines 5 and 9). These requests are independent and should not abort each other in `Timer`. Hence we state the following observation:

Observation 4 (No abort when command originates from same reaction) *Two commands or requests that originate from the same source reaction should not abort each other.*

However, we still want commands and requests to abort in all the other cases, thus leading to the following observation:

Observation 5 (Abort when newer command has same source connection) *Two commands or requests originating from the same source connection but different source reactions should abort each other. Commands or requests originating from newer source reactions should abort reactions originating from older source reactions.*

This observation is illustrated in Figure 12. When `window` sends `opened`, `VentilationMonitor` starts reaction 1 with two timers in parallel. Starting the two timers leads to two reactions, 2 and 3, in the `Timer`. If `window` sends `closed` before the timers are out, `window`'s `closed` will abort reaction 1 and start reaction 4. Reactions 2 and 3 will also abort because `timer_start` sent in reaction 4 aborts them and starts reaction 5.

6.2.5 Abort Semantics Rule

In the observations in the previous sections, we have distinguished between incoming notifications/solicits and commands/requests. The former arrive on the connections that a composition sets up, and the latter arrive to the synthesized services of a composition.

If we do not make this distinction, we can generalize observations 3, 4, and 5 to discuss incoming spontaneous messages in general, instead of only commands and requests. We can then see that observations 1 and 2 (which are about notifications and solicits) are special cases of the generalized observations 5 and 3, respectively. Furthermore, we can note that all possible cases are covered by the generalized observations 3, 4, and 5, with 3 and 4 (no abort) being complementary to 5 (abort). This allows us to reformulate observation 5 as a general rule for our abort semantics:

Rule 1 (Abort semantics)
A new reaction r' will abort an existing reaction r, if and only if reactions r and r' have the same source connection, but different source reactions, and where the source reaction of r' is newer than that of r.

6.3 Epochs for Implementing Abort Semantics

In order to implement abort semantics according to Rule 1, we will introduce the notion of *epochs*. We think of an epoch as having a place and a time, like the middle ages (Europe, 476–1453). In our case, the place is a source connection, and the time uses a logical clock (inspired by Lamport [Lam78]) representing the lifetime of a source reaction. An epoch represents the activity in the system caused directly or indirectly by a source reaction. Two epochs for the same source connection are ordered in time, and the idea is that a newer epoch aborts an older epoch for the same source connection. Epochs for different source connections are not comparable.

More formally, we represent an epoch as a tuple $(sourceId, seqNbr)$, where $sourceId$ is a universally unique id for the source connection, and $seqNbr$ is the logical time at which the source reaction started.

Existing native services in PALCOM are not aware of epochs, so when a source service sends a notification or solicit to a composition, the message contains no epoch information. Instead, it is the composition that is responsible for creating new epochs. This is done by associating a counter, $seqNbr$, with each source connection, keeping track of the starting time of the most recently created source reaction for that connection. When the composition receives a notification or a solicit, it increases the corresponding $seqNbr$ counter, and creates a new reaction with the epoch $(sourceId, seqNbr)$, where $sourceId$ is the universal id of the connection. When a composition is started, a universally unique $sourceId$ is created for each connection, based on the device id where the composition is running.

To determine if one epoch aborts another, we introduce the `aborts` relation. Given two epochs $e = (sourceId, seqNbr)$ and $e' = (sourceId', seqNbr')$, e' will abort e if they have the same $sourceId$ and if the $seqNbr$ of e' is strictly greater than that of e. Formally:

$$e' \texttt{ aborts } e \textbf{ iff } sourceId = sourceId' \wedge seqNbr' > seqNbr$$

After creating a new source reaction with an epoch e', the composition interpreter aborts any blocked reaction with an epoch e that e' `aborts`.

When a reaction sends commands and requests, the interpreter adds the reaction's epoch to the messages. Furthermore, when a composition receives a command or request on a synthesized service, the new reaction gets the epoch e' of the received message, and all blocked reactions with an epoch e are aborted, if e' `aborts` e.

In case the opposite holds, i.e., there is a blocked reaction with an epoch e that `aborts` e', then the new reaction is not created or run at all. We can view this as the new reaction being immediately aborted.

Note that a received command or request always has an epoch because these messages are received on a synthesized service, and are therefore sent by another composition.

In Figure 13, we use the window monitoring scenario to show how one new reaction can abort two blocked reactions in the same composition. The connection from `VentilationMonitor` to `window` has the universal id `e3b0`. Each message arriving on this connection gives rise to a new reaction, aborting the previous one, and starting a new epoch. Reaction 1 sends two messages to the `Timer`, leading to two new reactions, 2 and 3. Since they belong to the same epoch $(\texttt{e3b0}, 1)$, they will both be created without one aborting the other. The new reaction in `VentilationMonitor` has a newer epoch, $(\texttt{e3b0}, 2)$, so when it sends the message `timer_start(24h)`, a new reaction (5) is created at `Timer` and aborts the two earlier reactions 2 and 3, since $(\texttt{e3b0}, 2)$ `aborts` $(\texttt{e3b0}, 1)$.

6.3.1 Epoch-aware Native Services

While legacy PALCOM services are not aware of epochs, it is possible for new native services to make use of them. If a composition receives a message with an epoch from a native service, it will use this epoch for the new reaction, and abort earlier reactions as discussed above. Epoch-aware services are useful in order to implement strategy services, as will be discussed in the next section. Epoch-aware services are also useful when a service is connected to from two different compositions, and message flow goes from one composition through a service to another composition. This will be discussed in Section 8.

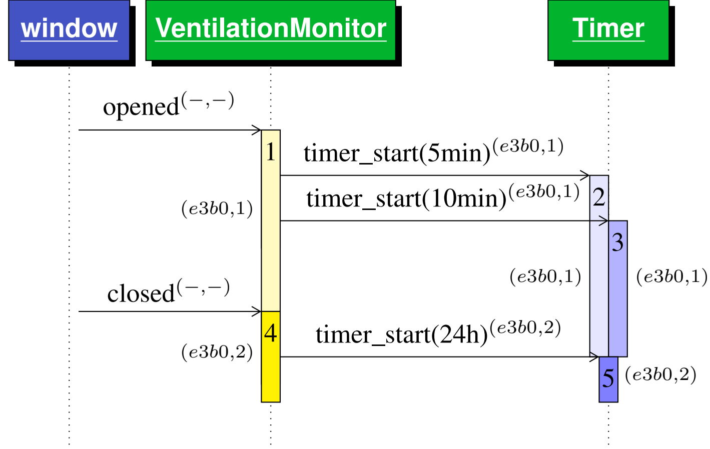


Figure 13: Epochs are shown on messages and reactions. `e3b0` is the universal id for the connection between `VentilationMonitor` and `window`. The message `timer_start(24h)` leads to a new reaction (5) that aborts reactions 2 and 3.

6.4 Bounded Number of Epochs

In this section, we show that the abort strategy limits the simultaneously ongoing activity in a system by bounding the number of epochs in it. We show this boundedness for systems without epoch-aware native services. To do this, we first show that a system has a bounded number of $sourceId$s, then that the number of epochs in a running composition is bounded by the number of $sourceId$s in the system. Lastly, we combine these two facts and show that the number of epochs in a system is bounded. Note that we here only consider static systems, i.e., systems where services and compositions are not added or removed.

To show that a system has a bounded number of $sourceId$s, we first note that the number of $sourceId$s equals the number of source connections. We also note that every source connection is defined in the connection part of the composition (Section 4.1); hence, a composition has a bounded number of source connections. We then get that the the number of source connections is bounded because every composition has a bounded number of source connections, and a system has a bounded number of compositions.

We now want to show that the number of $sourceId$s bounds the number of epochs in a running composition. All epochs with a given $sourceId$ in a running composition, have the same $seqNbr$ because the epoch with the highest $seqNbr$ for the given $sourceId$ aborts all reactions with the same $sourceId$ and a lower $seqNbr$. Thus, the number of epochs in a composition can never exceed the number of $sourceId$s in the system.

Combining these two facts, we get that the number of epochs in a running

composition is bounded. With the bounded number of epochs in a running composition and the bounded number of running compositions in the system, we get that the total number of epochs in the system is bounded.

It would be desirable to put a bound also on the number of reactions in a system. This would require that each epoch only can lead to a bounded number of reactions. Currently, we cannot guarantee this, because it is possible to construct message loops. A simple example is a composition that sends a command to a synthesized service, that in turn sends a notification back to the composition, which then sends the same command to the synthesized service again. This can lead to an infinite number of reactions, corresponding to an infinite loop. It would also be useful to determine boundedness of systems with *epoch-aware* native services. As future work, we plan to look into how both these problems can be solved by introducing more expressive service APIs.

In this section, we have informally shown that the number of epochs in a system is bounded. In the future, we may formalize the dynamic semantics to prove this formally, as well as extending the work to reason formally about when the number of reactions is guaranteed to be bounded.

7 Adapting Semantics with Strategy Services

In Section 6.1 we discussed the *parallel*, *queue*, *ignore*, and *abort* strategies for dealing with incoming messages. For COMPOS, we have chosen abort semantics, but sometimes, another strategy would be preferable. In this section, we will sketch how the other strategies can be accomplished by adding an extra native service, a *strategy service*.

7.1 Ignoring Messages

We will start by considering the *ignore* strategy, where incoming messages are ignored until the composition has completed an ongoing reaction.

As an example, consider the bird-watching system from Section 3. If a second `move` message arrives before the reaction for the previous `move` has completed, COMPOS will abort the ongoing reaction, as shown in Figure 14. If the motion sensor sends move messages with a sufficiently high frequency, no reaction will be able to finish. A more desirable strategy in this case would be to ignore incoming `move` messages while there is an ongoing reaction.

To implement the ignore strategy, we can use a strategy service called `latch` which implements the state machine shown in Figure 15. The latch is initially open. If it then receives a `signal` command, it will close and send out a `signaled` solicit. After that, it will ignore all incoming `signal` commands until it receives a `reset` response, at which it goes back to the open state.

Figure 16 shows an overview of the bird-watching system with the added `latch` service. The `latch` in effect breaks the message flow into two parts: the

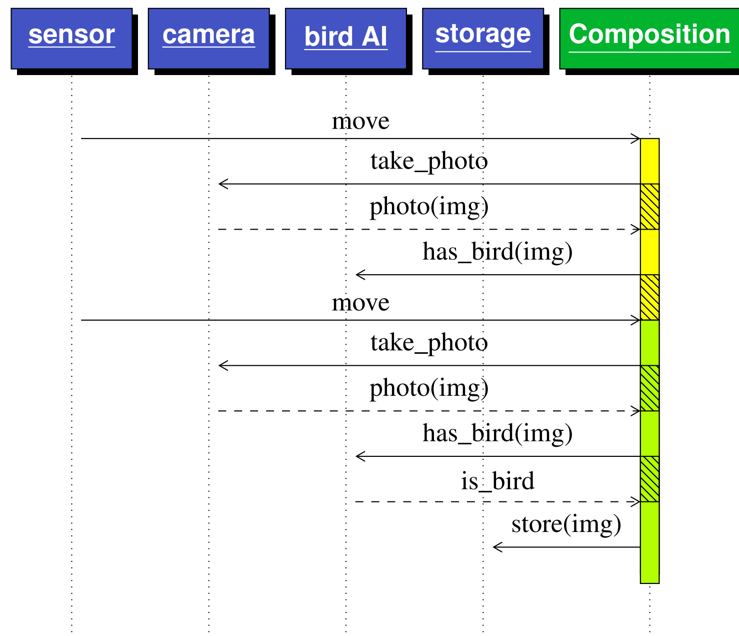


Figure 14: The sequence diagram shows the `move` notification aborting the currently blocked reaction (yellow) and initiating a new one (green).

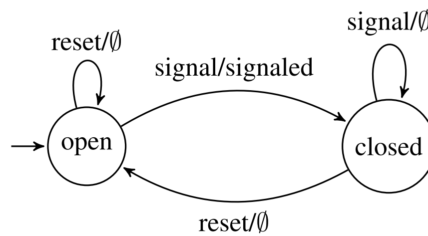


Figure 15: State machine for the `latch` service. Edges are labelled with (received/sent message) where ∅ means sending no message.

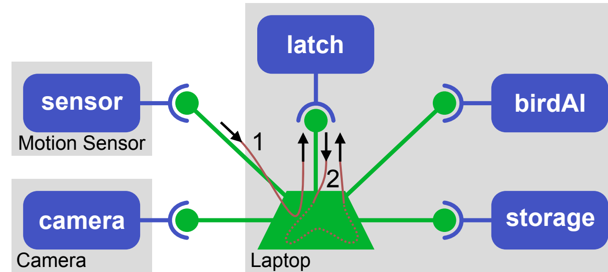


Figure 16: Bird-watching system extended with a `latch` strategy service that breaks the message flow into two separate parts.

first part goes from the `sensor` through the composition to the `latch` (flow 1), and the second part goes from the `latch` through the composition to the `camera`, the `birdAI` and the `storage` (flow 2). Because the two flows have separate sources, new epochs along flow 1 (due to new `move` messages) cannot abort ongoing reactions in flow 2.

In more detail, flow 1 starts when a `move` notification arrives from the sensor. The composition then sends a `signal` command to the `latch`. If the latch is open, it closes and starts flow 2 by sending a `signaled` solicit back to the composition. However, if the latch is closed, it simply ignores the incoming `signal` message. When the composition has finished handling a `signaled`, it responds with `reset` to the `latch` to open it again.

To use the `latch` service in the composition, a *when-do* is added that sends a `signal` to the `latch` every time a `move` is received. Further, the original *when-do* is changed to listen for the `signaled` solicit instead of the `move`, and to send a `reset` response at the end of the reaction. The resulting script is shown below:

```
1  when notif move from sensor do
2    send cmd signal to latch
3  when sol signaled from latch do
4    send req take_photo to camera
5    receive resp photo(var img) from camera
6    send req has_bird(img) to birdAI
7    select
8      when resp is_bird from birdAI do
9        send cmd store_image(img) to storage
10     when resp is_not_bird from birdAI do
11   end
12   send resp reset to latch
```

The change of the composition allows a reaction to run to completion, without being aborted by any move message. Figure 17 shows a scenario where the latch pattern prevents the abortion of a reaction.

By adding the `latch`, there is a risk that the reaction in the composition gets

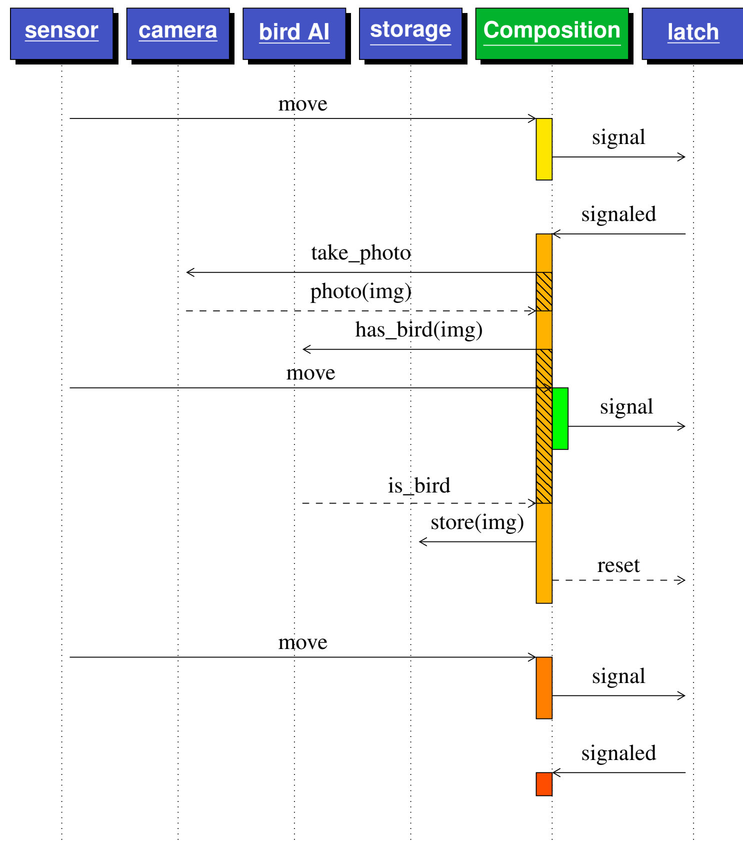


Figure 17: The sequence diagram shows the use of a latch service. When the first `signal` command arrives at the `latch`, it sends a `signaled` solicit back to the composition. However, when the second `signal` arrives, the `latch` ignores it. When the third `signal` arrives at the `latch`, it has recently received a `reset` response, and a `signaled` solicit is again sent to the composition. The use of the latch ensures that the reaction dealing with the photo always finishes.

stuck forever. For example, if the `birdAI` is on a separate device and the connection goes down, the reaction will be stuck waiting for a response that never arrives. To handle this situation, a timeout can be added. One possibility is to add the timeout directly to the `latch` service, so that after entering the closed state, the latch will automatically go back to the open state after a certain time. At the next `move` message, the `latch` will then send a `signaled` to the composition that will abort the ongoing reaction in the composition. Instead of placing the timeout behavior in the `latch` service, it is also possible put a time out in the composition using a timeout service.

7.2 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 `queue` service instead of a `latch`. The `queue` is like the `latch` but instead of ignoring messages it puts them in a queue. When it receives a reset it sends the next message in the queue.

Similar to the `latch`, the queue can be complemented with a timeout, to abort stuck reactions that wait for replies that have gone missing.

7.3 Parallelizing Messages

An alternative strategy can be to handle incoming notifications in parallel. For example, if the motion sensor sends a burst of move messages, we might like to start several reactions in parallel in order to take a number of camera shots as soon possible, and not have to wait for each image to be processed by the `birdAI` service.

To implement this strategy, we can use a `parallelizer` service. This service makes use of epochs to avoid that reactions are aborted. When the `parallelizer` service receives a `signal` command, it sends an `signaled` notification. In contrast to other native services we have discussed, the `parallelizer` service attaches an epoch to the notification. By using the same epoch for each `signaled` notification, a new reaction will be started in the receiving composition, without aborting previous reactions.

To use the `parallelizer` in a composition, we can add a *when-do* in a similar way as for the `latch`. The *when-do* listens for the message that we want to parallelize, and sends an `signal` command to the `parallelizer`. A separate *when-do* listens to the `signaled` message from the parallelizer, and performs the desired actions. Figure 18 shows a sequence diagram of the use of a `parallelizer` for the bird watching scenario.

By adding the parallelizer service, we get two new problems. First, there may be a system overload if the motion sensor generates very many messages. For this reason, we might like to limit the number of ongoing parallel reactions to a

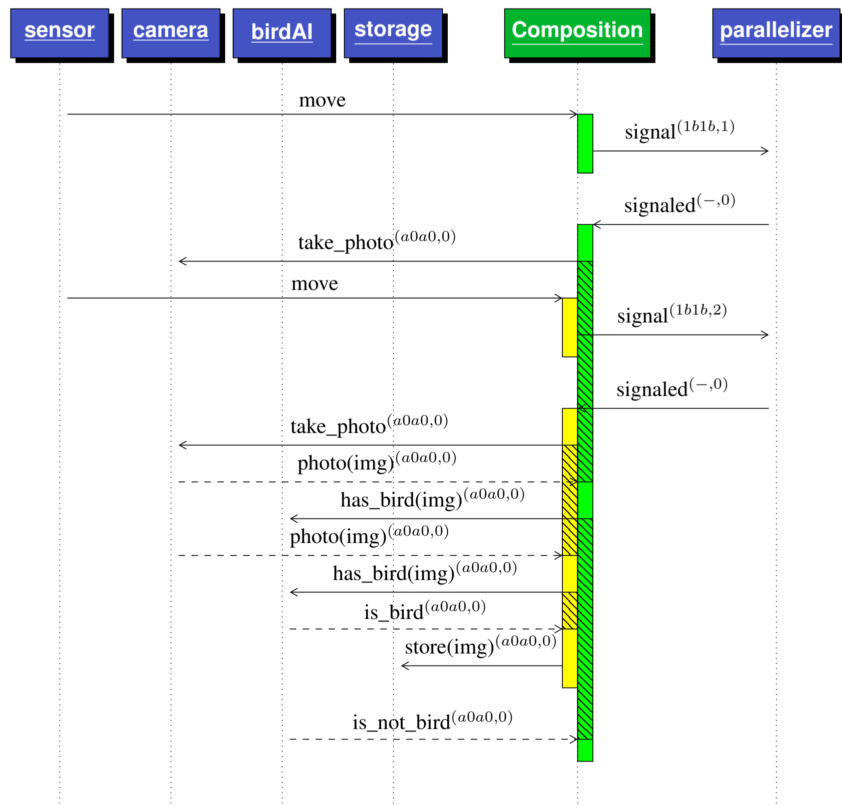


Figure 18: The sequence diagram shows the use of a parallelizer service. `a0a0` and `1b1b` are the universal ids for the connections from the composition to the sensor and the parallelizer service, respectively. The parallelizer adds the same epoch to all `signaled` messages, so that the reactions in the composition are not aborted.

maximum number, N. Second, since the reactions are not aborted, they might get stuck like for the latch and the queue. These problems can be solved by letting the parallelizer service use N internal latches with timeouts, and introduce a unique *sourceId* for each latch. By using different *sourceIds*, messages from the different latches will not abort each other. This more advanced parallelizer would route an incoming `signal` message to an open latch, or ignore it if all latches are closed. When a latch is closed, it sends a solicit `signaled`, using its *sourceId* in the epoch, waiting for a `reset` as a response. A latch is reset automatically if no response arrives within a given time, and the epoch *seqNbr* is increased for each reset. This way, if a latch opens after a timeout, the next message will abort a stuck reaction in the composition.

7.4 Discussion

In deciding the semantics for incoming spontaneous messages, we chose *abort* semantics as the default. This allows us to implement alternative strategies like *ignore*, *queue*, and *parallel* as strategy services. We also sketched how timeouts can be used for such strategy services to avoid that reactions get stuck if messages are lost, and how epochs can be used to further control the behavior. Note that we, so far in practice, only have tested the strategy services without timeouts.

In our examples above, the original message (`move`) does not carry any arguments. If we would like to add strategies to messages with arguments, we would need to supply a custom strategy service. Future work could look into supporting some kind of genericity so that parts of the API can be parameterized. Future work could also look into encapsulating the behavior modification in a separate strategy composition that exposes the modified message on its synthesized service. This way, the user composition would not have to handle the message flow to the strategy service explicitly, but could just connect to the strategy composition instead of to the original service. Genericity would be useful also in this case, in order to construct compositions that can be parameterized to different services and message types. Adding syntactic sugar to allow the end user to express different patterns concisely is also a possibility.

8 Evaluation

COMPOS is a superset of the original stateless composition language used in PALCOM. The main motivation for designing COMPOS was to enable more expressive compositions, capturing more complex message sequencing in an explicit way, yet keeping the language simple.

In order to evaluate COMPOS, we were interested in finding out if and how existing PALCOM applications can be improved by taking advantage of the new constructs. For this purpose, we conducted a case study, reimplementing a commercial e-health application, which we will discuss in Section 8.1.

We were also interested in finding out if COMPOS would be suitable for popular application areas like home automation. We have developed our own home automation scenarios for illustrating the different constructs in COMPOS, but would the language be useful also for scenarios constructed by others? To evaluate this aspect, we sketched reimplementations of typical home automation scenarios as identified by Rodríguez-Avila, de Koster, and de Meuter [Rdd21], which we will discuss in Section 8.2.

8.1 Case Study in E-health

As a case study, we reimplemented a scaled-down version of a commercial e-health system using COMPOS. The original system was built using PALCOM and its stateless composition language called the *assembly language*[5]. The system supports weight monitoring of patients with kidney failure and under peritoneal dialysis. For these patients, rapid weight changes can indicate the need for intervention from the caregiver.

Each patient has a tablet and a smart scale at home. The patients weigh themselves regularly on the scale, and the measurements are sent automatically, via the tablet, to servers at the hospital. The caregivers can access the data via a web interface. The tablet has an interface where the patient can chat with the caregiver and send pictures. The tablet interface also shows the weight history, depicted in Figure 19.

We conducted the case study together with the system designer responsible for the tablet. He scaled down the system for us, omitting some of the functionality, e.g., the authentication. We then reimplemented all the 11 compositions running on the tablet in COMPOS. After that, we compared the COMPOS reimplementations with the original implementations in the assembly language. The system designer also checked that our reimplementation seemed correct.

8.1.1 Architecture

Figure 20 shows the architecture of the e-health system.

We will not discuss the whole architecture in detail, but only point out a few important aspects of it:

User interface. In addition to devices, services, and compositions, the architecture includes a user interface (UI). UIs in PALCOM are built using a special tools [JM20] that allows developers to connect services to graphical interactive components. The UI in Figure 19 is built using these tools. A UI is similar to a composition in that it can set up connections to services. In the e-health system, the UI sets up connections to most other services running on the tablet.

[5] An *assembly* here refers to a set of connected devices, and is not related to assembly languages for processors.

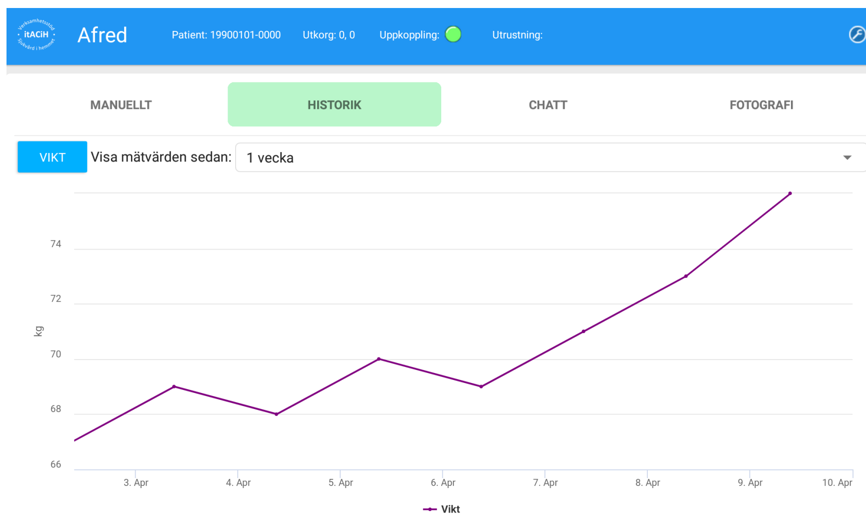


Figure 19: Screenshot (in Swedish) from a patient tablet showing the history of weight measurements.

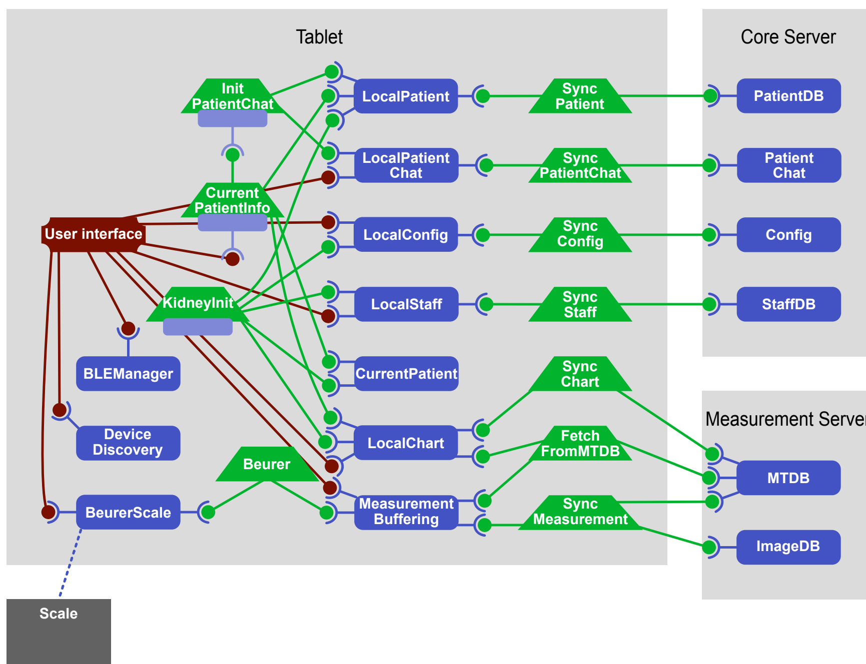


Figure 20: System architecture of the e-health scenario.

Synchronizing compositions. Many of the compositions simply synchronize information between the Tablet and the servers. For example, the composition `SyncPatient` synchronizes patient data between the `LocalPatient` service on the Tablet and a patient database (`PatientDB`) on the Core Server.

Bridges. External devices not speaking the PALCOM protocol are attached using bridge services. This is the case for the scale, which only communicates using a device-specific protocol over Bluetooth. The bridge to the scale is implemented by the service `BeurerScale` (Beurer is the brand of the scale).

Buffering. The measurements from the Scale are transmitted to the Measurement Server, but are buffered on the Tablet, since the connection between the Tablet and the Measurement Server might not be up all the time. The buffering is handled by the service `MeasurementBuffering`. The UI also connects to the `MeasurementBuffering` service in order to show if the current measurement has been synchronized with the server yet or not. Several other services use a similar pattern of buffering for the servers and providing information to the UI.

Care ID. For privacy reasons, all patient-specific data that is stored in server databases, like weights, chat histories, etc., are associated with a *Care ID*, a pseudonym for the patient, rather than with the patient name and/or civic number. The Care ID number is stored on the tablet by the `CurrentPatient` service. On the Core Server, the `PatientDB` service stores the mapping between Care IDs, patient names, and civic numbers, and all the other services only store data connected with Care IDs.

Separate servers. As a security measure, the system separates patients' medical data and the rest of the system by using two servers: the Measurement Server for medical data and the Core Server for everything else. According to one of the system designers, they also plan to put the `PatientDB` on its own separate server.

Aggregating compositions. Some compositions aggregate functionality from several services to provide information to the UI through a synthesized service. The `CurrentPatientInfo` composition is an example of this. It uses the Care ID from the `CurrentPatient` service in order to lookup the actual name and civic number of the patient so that they can be presented in the UI.

Authentication. The `KidneyInit` composition initializes a tablet to work for a specific patient. In the real system it is triggered from an authentication part of the system, but that is removed in our scaled-down version of the system. For testing purposes, we can trigger this composition manually in our reimplementation (via the PALCOM browser).

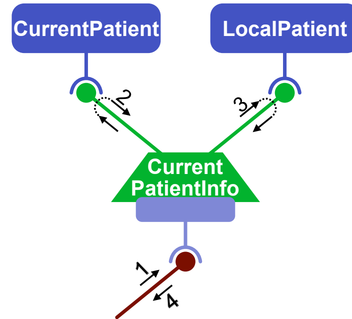


Figure 21: Message flow for the `CurrentPatientInfo` composition: 1) The UI asks for patient information. 2) The composition asks for the Care ID of the current patient. 3) The composition asks for patient information, using the Care ID as the key. 4) The composition responds to the UI with the patient information.

8.1.2 Reimplementation in COMPOS

In reimplementing the e-health system, we kept the original UI and all the native services, but replaced all the compositions on the tablet by COMPOS compositions. Since COMPOS is a superset of the original PALCOM assembly language, it is straightforward to replace the assembly compositions by COMPOS compositions that use the same stateless message flow. However, we have instead tried to design the new compositions so that they take advantage of the stateful constructs in COMPOS as much as possible.

One of the cases where COMPOS makes a difference is the `CurrentPatientInfo` composition. When the composition receives a request to its synthesized service, it asks the `CurrentPatient` service for the Care ID, and then uses the Care ID to look up various information for the patient, like the name and civic id which it gets from the `LocalPatient` service. If the information is missing, the `LocalPatient` replies with `no_patient_info`. Figure 21 shows the message flow.

In the original stateless language, this behavior had to be implemented as four different *when-dos*, treating each incoming message separately, with no apparent connection between the different messages. See Listing 1. The reimplementation, on the other hand, makes use of explicit sequences and alternatives using request-responses. There is now only one outer *when-do*, and the dependencies between different messages are explicit. See Listing 2.

Another example of taking advantage of COMPOS is for the synchronizing compositions. They all have a similar structure, forwarding notifications and solicit/responses between the two sides (tablets and servers). For example, consider the `SyncPatientChat` composition. It synchronizes the chat stored on the server (`PatientChat`) with the local buffer on the tablet (`LocalPatientChat`). In one

Listing 1: Original part of `CurrentPatientInfo` composition.

```
1  when req get_patient to CurrentPatientInfo do
2    send req get_current_patient to current_patient
3  when resp current_patient(var care_id) from current_patient do
4    send req get_patient(care_id) to local_patient
5  when resp patient_info(var name, var civic_id) from local_patient do
6    send resp patient_info(name, civic_id) from CurrentPatientInfo
7  when resp no_patient_info from local_patient do
8    send resp no_patient_info from CurrentPatientInfo
```

Listing 2: Refactored part of `CurrentPatientInfo` composition.

```
1  when req get_patient to CurrentPatientInfo do
2    send req get_current_patient to current_patient
3    receive resp current_patient(var care_Id) from current_patient
4    send req get_patient(care_id) to local_patient
5    select
6      when resp patient_info(var name, var civic_id)
7          from local_patient do
8        send resp patient_info(name, civic_id) from CurrentPatientInfo
9      when resp no_patient_info from local_patient do
10       send resp no_patient_info from CurrentPatientInfo
11   end
```

part of this synchronization, the service on the tablet sends a solicit and waits for a response. The composition forwards the solicit as a request to the server and when the response arrives, it is forwarded back to the tablet service.

In the original composition, this message flow has to be handled with two independent *when-dos*, so the message flow is not explicit, see listing 3. In the refactored composition, we can make the flow explicit with a single *when-do*, see listing 4.

8.1.3 Adapting to Epoch-unaware Services

A problem we encountered when doing the reimplementation is that neither the UI nor the services are epoch-aware. All their messages are sent without epoch

Listing 3: Original part of `SyncPatientChat` composition

```
1  when notif get_chat(var req_data) from local_patient_chat do
2    send cmd get_chat(req_data) to patient_chat
3  when notif chat(var resp_data) from patient_chat do
4    send cmd chat(resp_data) to local_patient_chat
```

Listing 4: Refactored part of SyncPatientChat composition

```
1 when sol get_chat(var req_data) from local_patient_chat do
2   send req get_chat(req_data) to patient_chat
3   receive resp chat(var resp_data) from patient_chat
4   send resp chat(resp_data) to local_patient_chat
```

numbers, and the receiving COMPOS composition will then attach a new epoch to each incoming message.

It turned out that the UI (which is similar to a composition) sends two requests to the CurrentPatientInfo composition and waits for the responses to arrive in any order. The UI is thus implemented using a message flow similar to the parallel construct in COMPOS. If no special measures are taken, these requests will abort each other. We solved this by placing a strategy service between the UI and the CurrentPatientInfo so that the requests seem to come from different sources. A preferable solution would be to make the UI epoch-aware and let it attach epochs to its outgoing messages. Since it sends out two requests in parallel, it should use the same epoch for both of them.

A similar situation occurred with some of the local buffering services. A service may send two solicits on the same connection, and then wait for the responses to arrive in any order. Again, the composition will attach different epochs to these solicits so they abort each other. For these cases, we solved the issue by adding a *queue* strategy service. This could be avoided if we reimplemented the services to be epoch-aware.

8.1.4 Conclusions Case Study

Because the assembly language is stateless, the system's services operate such that the compositions can be stateless. Hence, we may not fully utilize the stateful features of COMPOS in this implementation of the system. In contrast to the assembly language, we designed COMPOS to allow for state in the compositions. If the system was designed using COMPOS from the start, we think the composition would use more COMPOS constructs. Despite the system being designed for stateless reactions, our reimplementation uses stateful features not present in the assembly language. We see this as an indication that these stateful features can be helpful even when migrating systems from the assembly language to COMPOS.

One advantage we see of using COMPOS is the explicit control flow of the messages, e.g., the syntactic separation of responses and notifications. In [Å+20], we use this explicit control flow in analyses of PALCOM systems. If we would like to extract control flow from compositions written in the original assembly language, we would need to know how native services operate to extract their internal control flow. We hypothesize that explicit control flow also makes compositions easier to understand for humans, but this is future work to investigate.

8.2 Smart Home Scenarios

Rodríguez-Avila, de Koster, and de Meuter [Rdd21] have identified seven smart home scenarios that involve multiple messages from multiple devices. These scenarios are complex to express using ordinary distributed actor systems that only match a single message at a time, since such systems need to introduce state variables to keep track of when different combinations of messages have been seen. Rodríguez-Avila et al. show how to elegantly encode these scenarios in *Sparrow*, a declarative DSL that can act based on complex multiple-message matching. By polling four online forums, they show that the scenarios represent patterns occuring in real-world home automation systems. Quoting directly from their paper, these are the scenarios:

> 1. Turn on the lights in a room if someone enters, and the ambient light is less than 40 lux.
> 2. Turn off the lights in a room after two minutes without detecting any movement.
> 3. Send a notification when a window has been open for over an hour.
> 4. Send a notification if someone presses the doorbell, but only if no notification was already sent in the past 30 seconds.
> 5. Detect home arrival or leaving based on a particular sequence of messages, and activate the corresponding scene.
> 6. Send a notification if the combined electricity consumption of the past three weeks is greater than 200 kWh.
> 7. Send a notification if the boiler fires three Floor Heating Failures and one Internal Failure within the past hour, but only if no notification was sent in the past hour.

To evaluate how well COMPOS works for these scenarios, we have sketched COMPOS solutions for them (see appendix B). We found that we could write all of the scenarios in a fairly straightforward way, and without using explicit state variables. However, we should mention that we have so far not tested our solutions in PALCOM, due to lack of time. Our solutions are not as declarative as those in Sparrow, but we find most of the scenarios quite straightforward to express, by relying on reaction state like sequencing, parallel, and finish-first, on the abort semantics, and on strategy services like latches. An additional difference is that we use external native services for time-outs and computations, but Sparrow has built-in constructs for this which contributes to making their solutions more concise.

Scenario 3, also discussed in Section 6.2 is quite typical of how these scenarios can be written in COMPOS in a straightforward way, and which uses the abort semantics for resetting the timer. Scenario 7 is the most complex scenario. Here,

our lack of built-in computation support gives a fairly lengthy solution which may be a bit cumbersome to read. The solution has three extra services: a timer, a counter, and a service to compare two numbers. In principle, we could get a shorter solution by combining functionality of the extra services into a single service, but that would make the new extra service less general.

Our goal has not been to provide the most concise solutions, but rather to provide a simple and yet expressive composition language. In comparing to Sparrow, that was particularly designed to support multi-message patterns, we can note that Sparrow does not support request-response. Thus, their solution to scenario 1 needs to listen for constantly emitted messages about the ambient light. In contrast, our solution can request the luminosity level from the ambient light sensor.

In our implementation, an ongoing COMPOS reaction can only wait for responses, and not for notifications or commands. By relaxing this requirement, we would be able to simplify some of the scenarios. For example, in scenario 5, our solution sends a request for the next `open` response from the door. By a relaxed definition, we would be able to remove the request and simply let the reaction wait for the next `open` notification.

9 Related Work

There are many different kinds of related work for COMPOS. We focus on briefly discussing previous composition languages for PALCOM, composition for web services, end-user development for IoT, other actor-based approaches, and reactive programming.

9.1 Previous Composition Languages for PALCOM

Svensson, Hedin, and Magnusson introduced compositions and synthesized services in the PALCOM *assembly language* [SHM07]. These compositions are stateless, limiting reactions to only contain actions for sending messages and setting global variables. The current PALCOM release (4.0.19)[6] furthermore supports that messages can be interpreted as requests or responses by piggybacking reply information into the messages. This allows a service to reply to the correct sender, in case several compositions connect to the same service. However, compositions are still stateless, and replies are handled in the outermost event loop of a composition, just like other messages.

A previous experiment of creating more advanced compositions for PALCOM was done by Linus Åkesson [Åke16]. This language was similar to COMPOS in that compositions had state and supported nested and parallel action sequences. It differed in that it used the parallel strategy as the default semantics, and used timeouts to avoid indefinitely running reactions. The language was purely text-based,

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

without any integration with a GUI or the PALCOM browser, and was not intended for end users. The main goal of this language was to investigate how compositions could be automatically partitioned and distributed in order to minimize latency. This is an interesting line of research that might be applied also for COMPOS.

9.2 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 only.

9.3 AmbientTalk

AmbiantTalk [Cut+07] is a domain-specific language developed for programming applications in mobile ad-hoc networks. AmbiantTalk is an object-oriented and actor-based language that supports sending messages between actors on different nodes. AmbiantTalk supports unreliable connections by buffering messages while the connection is down. By using a leasing scheme, AmbiantTalk does not buffer messages infinitely. A connection renews the lease every time it sends or receives a message, and if the lease expires, it removes the buffered messages and raises exceptions.

Like AmbientTalk, COMPOS is also designed for programming applications in mobile ad hoc networks. However, unlike COMPOS, AmbiantTalk targets developers and does not separate services and compositions. The lease model and the use of exceptions also differ from the "best-effort" model used in COMPOS, where weak connectivity is regarded as normal.

9.4 End-user Development for IoT

There are different approaches for end-user development of IoT systems. Some use programming by demonstration [Li+17], whereas others use different types of DSLs, like TeC [Sou+11], Midgar [Gar+14], 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 out-

put events of activities, but do not themselves contain any event logic or message adaptation.

Midgar [Gar+14] 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-do*s 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.

9.5 Trigger-action Programming

A typical programming paradigm used for programming IoT systems is *trigger-action programming*. In this paradigm, a program consists of a trigger with an action. The trigger specifies an event that, when it occurs, executes the action, e.g., when the door opens, turn on the lights [Ur+14]. Trigger-action programming is currently used in commercial systems such as IFTTT[7]. The outer *when-do*s in COMPOS kan be viewed as a kind of trigger-action system, where the triggers are the outer `when`s, and the actions are the reaction specifications.

9.6 Reactive Programming

The reactive programming (RP) paradigm is a popular way of developing event-driven systems [Bai+13]. In RP, sources produce signals (e.g., motion events). These signals are combined using signal combinators to create new signals. The signal combinators, together with the sources, form a dependency graph for the system. So, when a source produces a new signal, it gets propagated through the dependency graph to sinks. Using RP, programmers can create interactive systems declaratively.

A specific kind of RP is Distributed Reactive Programming (DRP). DRP distributes sources, signal combinators, and sinks on different nodes in a network. If we compare our work to DRP, we can see that both combine sources, either using signal combinators or compositions. However, the signal combinators can do arbitrary computation and combine multiple signals originating from multiple sources.

[7] https://ifttt.com

In comparison, every reaction in COMPOS ties only to one source. To combine data from multiple nodes, COMPOS uses request/response.

An RP system needs a propagation algorithm to propagate the values correctly and efficiently through the dependency graph. In a DRP system, the propagation algorithm may need to be decentralized and handle unstable connections. Myter, Scholliers, and de Meuter [MSd19] proposes a propagation algorithm for DRP systems called QPROP. The signals in QPROP contain something similar to epochs called `sClocks`. `sClocks` is a dictionary where the keys are the sources, and the values are the logical time of the source data used to derive the signal. Whereas epochs are used to abort the right reactions, the QPROP algorithm provides glitch-free DRP, i.e., derived signals only depend on one signal value from each source.

10 Conclusions and Future Work

In this paper, we have presented COMPOS, a DSL for composing services into IoT systems. As a platform for our DSL, we use the PALCOM IoT middleware. COMPOS extends the existing composition language in PALCOM by adding stateful reactions (Sections 4 and 5). These stateful reactions allow compositions to wait for responses and to make use of parallel behavior.

With stateful reactions, we must have a strategy for removing reactions that might otherwise wait forever. In this paper, we have chosen to investigate and implement a strategy we call *abort* (Section 6). The basic idea behind abort is to remove old reactions when newer messages arrive. We introduced the notion of epochs to be able to treat reactions with the same origin together.

In some cases, the abort strategy is not desirable, and another strategy is more useful. We have shown how to change the strategy by adding a strategy service (Section 7).

To evaluate COMPOS with the abort strategy, we conducted a case study (Section 8). In the case study, we reimplemented 11 compositions in a scaled-down version of a commercial home care system built using PALCOM. We compared our implementation with the original and saw that the message flow is more explicit in COMPOS. The fact that the message flow is explicit has been used in Åkesson, Hedin, Fors, Schöne, and Mey [Å+20] to analyze IoT systems. We also solved seven home automation scenarios proposed by Rodríguez-Avila, de Koster, and de Meuter [Rdd21] using COMPOS.

In the future, we would like to investigate how COMPOS generalises by trying it in another IoT framework than PALCOM. To make it easier to change strategy, we want to add generic parameterization services and syntactic sugar for strategy services. Also, we would like to continue looking into the end-user programming perspective of COMPOS and do user studies to evaluate its usability. Another future direction is to formalize COMPOS to prove properties, such as that a system has a limited number of reactions or that refactorings do not change the system

function. We may also look into more complex message patterns with support for loops. Another track is to make our implementation more mature, and to use COMPOS in the e-health company already using PALCOM. We think that using COMPOS for building full production systems can give valuable suggestions for further improvements.

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. We thank Björn Johnsson and Mattias Nordahl for feedback on earlier drafts of this paper and Björn Johnsson for helping us with the case study.

References

[Åke+19] 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.

[Bai+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. “A Survey on Reactive Programming”. In: *ACM Comput. Surv.* 45.4 (Aug. 2013).

[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.

[Chr+01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C Note. World Wide Web Consortium, Mar. 2001.

[CC16] Joëlle Coutaz and James 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.

[Der+15] 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.

[Gar+14] 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.

[JM20] Björn A. Johnsson and Boris Magnusson. "Towards end-user development of graphical user interfaces for internet of things". In: *Future Generation Computer Systems* 107 (2020), pp. 670–680.

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), 558–565.

[LMS05] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC4122. Internet Engineering Task Force, July 2005.

[Li+17] 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.

[MSd19] Florian Myter, Christophe Scholliers, and Wolfgang de Meuter. "Distributed Reactive Programming for Reactive Distributed Systems". In: *Art Sci. Eng. Program.* 3.3 (2019), p. 5.

[Nor+20] Mattias Nordahl, Boris Magnusson, Görel Hedin, and Alfred Åkesson. "Smart bikes: Gradual update of IoT systems". In: *2020 IEEE 24th International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE. 2020, pp. 99–102.

[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.

[Rdd21] Humberto Rodríguez-Avila, Joeri de Koster, and Wolfgang de Meuter. “Advanced Join Patterns for the Actor Model based on CEP Techniques”. In: *Art Sci. Eng. Program.* 5.2 (2021), p. 10.

[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] David Svensson, Görel Hedin, and Boris 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.

[Ur+14] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. “Practical Trigger-Action Programming in the Smart Home”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’14. Toronto, Ontario, Canada: Association for Computing Machinery, 2014, 803–812.

[ÅH17] 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, 2017, pp. 7–12.

[Åke+18a] 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.

[Åke+18b] 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.

[Å+20] Alfred Åkesson, Görel Hedin, Niklas Fors, Rene Schöne, and Johannes Mey. "Runtime Modeling and Analysis of IoT Systems". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020.

[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 syntax for COMPOS used in this paper using ANTLR style EBNF[8].

```
composition : 'composition' name_def bindings
  synthesized_service* script;
bindings :  device_variable* service_variable*;
device_variable : 'device' name_def '=' device_ref;
service_variable : 'service' name_def '=' service_ref
  'on' device=name_use;
synthesized_service : 'synthesized service' name_def message_def*;
message_def :
   in_message
  | out_message
  | pattern;
in_message : 'in' name_def '(' parameter* ')';
out_message : 'out' name_def '(' parameter* ')';
parameter : name_def ':' type;
pattern : 'pattern' msg=name_use '->'
  '(' msg=name_use ( '|' msg=name_use )* ')';
script :  when_do_outer*;
when_do_outer :
    'when' 'cmd' msg=name_use ('(' ('var' name_def)* ')')?
      'to' service=name_use 'do' branch
  | 'when' 'req' msg=name_use ('(' ('var' name_def)* ')')?
      'to' service=name_use 'do' branch
  | 'when' 'notif' msg=name_use ('(' ('var' name_def)* ')')?
      'from' synth=name_use 'do' branch
  | 'when' 'sol' msg=name_use ('(' ('var' name_def)* ')')?
      'from' synth=name_use 'do' branch;
branch : action*;
action :
    receive
  | select
  | parallel
  | finish_first
  | send;
receive :
    'receive' 'resp' msg=name_use ('(' ('var' name_def)* ')')?
      'from' service=name_use
  | 'receive' 'resp' msg=name_use ('(' ('var' name_def)* ')')?
      'to' synth=name_use;
select : 'select' when_do_inner* 'end';
when_do_inner :
    'when' 'resp' msg=name_use ('(' ('var' name_def)* ')')?
      'from' service=name_use 'do' branch
```

[8] https://www.antlr.org

```
  | 'when' 'resp' msg=name_use ('(' ('var' name_def)* ')')?
      'to' synth=name_use 'do' branch;
parallel : 'parallel' (branch 'and')* branch 'end';
finish_first : 'finish first' (branch 'or')* branch 'end';
send :
    'send' 'cmd' msg=name_use('(' arg+ ')')?
      'to' service=name_use
  | 'send' 'resp' msg=name_use('(' arg+ ')')?
      'to' service=name_use
  | 'send' 'req' msg=name_use('(' arg+ ')')?
      'to' service=name_use
  | 'send' 'notif' msg=name_use('(' arg+ ')')?
      'from' synth=name_use
  | 'send' 'resp' msg=name_use('(' arg+ ')')?
      'from' synth=name_use
  | 'send' 'sol' msg=name_use('(' arg+ ')')?
      'from' synth=name_use;
arg : name_use | STRING;
```

To allow for comma-separated lists, we see comma as a token divider besides whitespace.

A.2 Semantics

In this section, we specify the COMPOS interpreter in pseudo-code. To only highlight the interesting features, we have omitted some functionality, such as variables.

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	sending-reaction-id of the sending service
sendId	send-id of the sending service
epoch	epoch from the sending service

Reaction

Field	*Description*
connection	connection to the service initiating the reaction
reactionId	the sending-reaction-id
epoch	the epoch of the reaction
remoteReactionId	the sending-reaction-id in the message initiating the reaction
remoteSendId	the send-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
seqNbrs := (*,0)                      ▷ map from sourceId to highest seqNbr, all initialized to zero
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 := new Reaction()
        reaction.currentAction := reactionSpec.startAction()
        reaction.connection := message.connection
        reaction.reactionId := reactionId
        reaction.remoteReactionId := message.reactionId
        reaction.remoteSendId := message.sendId
        if message.hasEpoch() then
            reaction.epoch = message.epoch
        else
            seqNbr := seqNbrs[reaction.connection]
            seqNbrs[reaction.connection] := seqNbr + 1
            reaction.epoch.seqNbr = seqNbr
            reaction.epoch.sourceId = reaction.connection
        end if
        UPDATEREACTIONSWITH(reaction)
        reactionId := reactionId + 1
    end if
    for reaction in reactions do
                  ▷ There will be at most one reaction that matches (currently not checked)
        RUNUNTILBLOCK(reaction, message)
    end for
end loop
```

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

```
procedure UPDATEREACTIONSWITH(reaction)
    reactions.add(reaction)
    for reactionLoop in reactions do
        epochLoop := reactionLoop.epoch
        if epochLoop.sourceId = epoch.sourceId then
            if epochLoop.seqNbr < epoch.seqNbr then
                reactions.remove(reactionLoop)
            else if epoch.seqNbr < epochLoop.seqNbr then
                reactions.remove(reaction)
            end if
        end if
    end for
end procedure

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

procedure (receive ∨ when_do_inner).MATCHRESPONSE(reaction, message)
    return (message.name = this.msg
           ∧ ( message.connection = this.service().connection
               ∨ message.toService = this.synth() )
           ∧ message.seqNbr = reaction.seqNbr
           ∧ message.reactionId = reaction.reactionId )
end procedure

procedure composition.MATCHSPONTANEOUS(message)
    for when_do_outer in this.allWhenDoOuter() do
        if  message.name = when_do_outer.msg
            ∧ ( message.connection = when_do_outer.service().connection
                ∨ message.toService = when_do_outer.synth())  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
        reaction.currentAction = this.nextAction()
        return TRUE
    end if
    return FALSE
end procedure

procedure select.PERFORM(reaction, message)
    for when_do_inner in this.when_do_inners do
        if when_do_inner.MATCHRESPONSE(reaction, message) then
            reaction.currentAction = when_do_inner.startAction()
            return TRUE
        end if
    end for
    return FALSE
end procedure

procedure send.PERFORM(reaction, message)
    if this is (cmd ∨ req) then
        send(this.service.connection, this.msg, this.args(...),
            reaction.reactionId, this.sendId(), reaction.epoch )
    else if this is resp ∧ this has to then                ▷ Respons to solicit
        send(this.service().connection, this.msg, this.args(...),
            reaction.remoteReactionId, reaction.remoteSendId, reaction.epoch)
    else if this is (notif ∨ sol) then
        for service in this.synth().connectedServices() do
            send(service.connection, this.msg, this.args(...), reaction.reactionId,
                this.sendId() ,reaction.epoch)
        end for
    else if this is resp ∧ this has from then              ▷ Respons to request
        send(reaction.connection, this.msg, this.args(...), reaction.remoteReactionId,
            reaction.remoteSendId, reaction.epoch)
    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.seqNbr := reaction.seqNbr
            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
```

Appendix B Home Automation Scenarios

Scenario 1 Turn on the lights in a room if someone enters, and the ambient light is less than 40 lux.

```
1  composition Scenario_1
2    ...
3    // Motion sensor in the room
4    service motion = ...
5    // Sensor measuring ambient light
6    service light_sensor = ...
7    // Computation service for comparing numbers
8    service compare = ...
9    when notif movement from motion do
10     send req get_illuminance to light_sensor
11     receive resp the_illuminance(var illum) from light_sensor
12     send req greater(illum, "40") to compare
13     select
14       when resp true from compare do
15           send cmd turn_on to light
16       when resp false from compare do
17     end
```

We may need to add a latch service (see Section 7) if the motion sensor sends movement notifications with such a high frequency that a reaction gets aborted before it has time to finish.

Scenario 2 Turn off the lights in a room after two minutes without detecting any movement.

```
1  composition Scenario_2
2    ...
3    // Motion sensor in the room
4    service motion = ...
5    // Light in the room
6    service light = ...
7    // Timer service for tracking time
8    service timer = ...
9    when notif movement from motion do
10     send req timer_start("2min") to timer
11     receive resp timer_end from timer
12     send cmd turn_off to light
```

This scenario relies on abort semantics. An ongoing reaction that is waiting for the timer will be aborted when a new movement happens.

Scenario 3 Send a notification when a window has been open for over an hour.

```
composition Scenario_3
  ...
  // Sensor on the window
  service window = ...
  // Timer service for tracking time
  service timer = ...
  // Service for sending notifications
  service alert = ...
  when notif opened from window do
   send req timer_start("1h") to timer
   receive resp timer_end from timer
   send cmd msg("Window open 1h") to alert
  when notif closed from window do
```

This scenario relies on abort semantics. An ongoing reaction that is waiting for the timer will be aborted when a new `opened` or `closed` happens.

Scenario 4 Send a notification if someone presses the doorbell, but only if no notification was not already sent in the past 30 seconds.

```
composition Scenario_4
  ...
  // Service for the doorbell
  service doorbell = ...
  // A type of strategy service
  service latch = ...
  // Timer service for tracking time
  service timer = ...
  // Service for sending notifications
  service alert = ...
when notif ring from doorbell do
  send cmd signal to latch
when sol signaled from latch do
  send cmd msg("ring") to alert
  send req timer_start("30s") to timer
  receive resp time_end from timer
  send resp reset to latch
```

The latch service is described in Section 7.

Scenario 5 Detect home arrival or leaving based on a particular sequence of messages, and activate the corresponding scene. More specifically: A person arriving home is detected by first detecting motion in front of the door, then that the door is opened, and finally that there is movement in the entrance hall, all within 60 seconds.

We only show the arrival scenario here. The leaving scenario is very similar.

```
1  composition Scenario_5
2    ...
3    // Motion sensor in front of the the door
4    service motion_door = ...
5    // Motion sensor in the entrance hall
6    service motion_entrance = ...
7    // Service for the door
8    service door = ...
9    // Timer service for tracking time
10   service timer = ...
11   // Service for sending notifications
12   service alert = ...
13 when notif movement from motion_door do
14   send req timer_start("60s") to timer
15   finish first
16     receive resp timer_end from timer
17   or
18     send req wait_open to door
19     receive resp open from door
20     send req wait_motion to motion_entrance
21     receive resp motion from motion_entrance
22     send cmd msg("home") to alert
23   end
```

In this implementation we assume that the `motion_entrance` and the `door` receive `wait` requests that respond when the respective event happens.

Scenario 6 Send a notification if the combined electricity consumption of the past three weeks is greater than 200 kWh.

```
1  composition Scenario_6
2    ...
3    // Service for the electricity meter
4    service meter = ...
5    // Service for storing and doing computation on time series
6    service time_series = ...
7    // Computation service for comparing numbers
8    service compare = ...
9    // Service for sending notifications
10   service alert = ...
11   when notif measurement(var amount) from meter do
12     send cmd add_measurement(amount) to time_series
13     send req sum_latest("3week") to time_series
14     receive resp result(var sum) from time_series
15     send req greater(sum, "200") to compare
16     select
17       when resp true from compare do
18         send cmd msg("High electricity consumption") to alert
19       when resp false from compare do
20     end
```

Scenario 7 Send a notification if the boiler fires three Floor Heating Failures and one Internal Failure within the past hour, but only if no notification was sent in the past hour.

In addition to the `boiler` (that can send failures) and an `alert` service (for sending notifications to), our solution makes use of an `event_counter`, a `comparer`, and a `timer` service. The `event_counter` can record events with time stamps, and can be requested to return the number of events of a specific kind that have occurred during the latest n minutes or hours. The `comparer` can compare numbers, for example using greater or equal. The `timer` service can be set and respond with a timeout. The `timer` can also be requested if it is currently `running`, i.e., if it has been set but not yet responded with a timeout.

```
1  composition Scenario_7
2    ...
3    service ...
4    when notif Floor_Heating_Failure from boiler do
5      send cmd count("floor heating failure") to event_counter
6
7      send req events_during("floor heating failure", "1h")
8        to event_counter
9      receive resp events(var nbr_floor) from event_counter
10
11     send req greatereq(nbr_floor, "3") to comparer
12     receive resp true from comparer
13
14     send req events_during("internal failure", "1h")
15       to event_counter
16     receive resp events(var nbr_Internal) from event_counter
17
18     send req greatereq(nbr_internal, "1") to comparer
19     receive resp true from comparer
20
21     send req running to timer
22     receive resp false from timer
23
24     send cmd msg("Boiler problems") to alert
25     send cmd start("1h") to timer
26
27   when notif Internal_Failure from boiler do
28     send cmd count("internal failure") to event_counter
29
30     send req events_during("floor heating failure", "1h")
31       to event_counter
32     receive resp events(var nbr_floor) from event_counter
33
34     send req greatereq(nbr_floor, "3") to comparer
35     receive resp true from comparer
36
37     send req running to timer
38     receive resp false from timer
39
40     send cmd msg("Boiler problems") to alert
41     send cmd start("1h") to timer
```

Note that if the `comparer` responds `false` (or if the `timer` responds `true`), there is no matching *receive*, and the reaction will be blocked until it is aborted. This gives shorter code than if we had used nested *select* statements.

PAPER III

Runtime Modeling and Analysis of IoT Systems

PAPER III

Abstract

Internet-of-things systems are difficult to understand and debug due to their distributed nature and weak connectivity. We address this problem by using relational reference attribute grammars to model and analyze IoT systems with unreachable parts. A transitive device-dependency analysis is given as an example.

1 Introduction

Internet-of-Things (IoT) systems are heterogeneous distributed systems that include embedded devices with sensors and actuators, as well as edge computers and/or servers. IoT systems can be even more difficult to understand and debug than ordinary distributed systems, since they may include mobile devices with weak connectivity. This results in a dynamic topology where devices are not always available [TM17].

One way of supporting better understandability and debugging is to compute a runtime model of the IoT system, and to support analysis of such a model [BBF09]. For example, in a smart home, this could allow the network of connected devices to be visualized. An example of an analysis could be to compute which devices need to work in order for the lights to turn on when the door is opened. Analyses

Alfred Åkesson, Görel Hedin, Niklas Fors, Rene Schöne, and Johannes Mey. "Runtime Modeling and Analysis of IoT Systems". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020

like these can be useful both for debugging and for finding problems when testing a system, before release.

For this approach to work well, the architecture has to be *explicit* in the sense that communicating components and connectors can be extracted from the system. In particular, it can be beneficial if the connectors have first class status [Sha93], and are not implicit in the code of the components. In particular, if a domain-specific language is used for specifying connectors between components, the architecture is easy to extract. Furthermore, it is desirable to specify analyses of the architecture in a high-level way, preferably using a declarative approach.

It is also important to note that an IoT system might not be defined by a single description in one location, but can emerge from many partial descriptions on different devices. The system may be constantly changing due to new components and connectors being added or removed. Furthermore, because of weak connectivity, the view of the system from any single device can be incomplete.

While most language and modeling approaches rely on conceptual modeling frameworks (e.g., EMF [Ste+08]) or runtime modeling frameworks (e.g., KMF [Fra+14]), these only explicitly support specific kinds of analysis, such as type analysis or constraint checking, and fall back on general-purpose languages for other kinds of analysis.

Thus, in this paper we propose using *Relational Reference Attribute Grammars* [Mey+20] for modeling and analyzing IoT systems with an explicit architecture. Reference Attribute Grammars (RAGs) support declarative analysis over abstract syntax trees and are used for building compilers and other language-based tools. Relational RAGs extend the abstract syntax of RAGs with relations, so that the structure to analyze is a conceptual model rather than an abstract syntax tree.

We evaluate the approach by developing a runtime model for PALCOM [SF+09], an IoT middleware toolkit that uses an explicit architecture. PALCOM components are called *services*, and connectors are called *compositions*. The compositions are described by DSL scripts that define what services to connect to, and how messages are mediated. The toolkit provides a discovery manager that keeps track of all connected devices, and what services and compositions that run on them.

As an example analysis, we have formulated and implemented a simple *device dependency analysis* (DDA). The DDA computes what devices need to be available and connected in order for a specific event to happen, such as turning on a light when a door opens.

Our contributions are the following:

- We present a basic runtime model for the PALCOM IoT architecture, formalized using Relational RAGs (Section 2).
- We present a home automation scenario as a motivating example (Section 3).

- We present an extended runtime model that can handle incomplete systems (where devices can be unavailable) and that includes composition scripts that enable more fine-grained analyses than the basic model (Section 4).
- We introduce and formalize the Device Dependency Analysis (DDA), and show how it can be specified using Relational RAGs on top of our extended runtime model (Section 5)

We end with related work and conclusions (Sections 6 and 7).

2 Basic Runtime Model

A running PALCOM IoT system consists of a set of devices running the PALCOM middleware, together with a number of service and composition instances, each hosted on a particular device. Each service provides an API of asynchronous messages that it can send and receive. However, a service does not set up any connections on its own. Instead, *compositions* act as connectors, each connecting to a number of services.

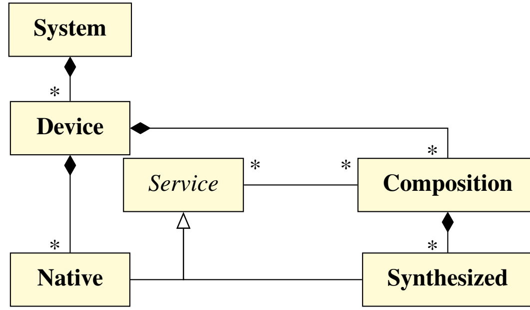
For this paper, all compositions are written in a domain-specific language called COMPOS [Åke+19]. A composition mediates between services by receiving and sending messages from/to them. A composition can also provide *synthesized* services that other compositions can connect to. Ordinary services (that are not synthesized) are called *native*, and are typically written in a general-purpose language like Java.

Because all connections are available in the composition scripts, which are easy to analyze, the complete component-connector architecture is available without having to analyze the general-purpose code of the native services.

Figure 1 shows the basic conceptual model of a running PALCOM IoT system, and the corresponding abstract grammar of the Relational RAG. The grammar specifies containment relations using grammar productions, and non-containment relations using the `rel` construct.

A Relational RAG can be extended with attributes and equations in order to declaratively specify derived properties. Each attribute of a node is defined by an equation, either in the node itself (labelled by ↑), or in an ancestor in the containment hierarchy (labelled by ↓). (In attribute grammar terminology, ↑/↓ attributes are called synthesized/inherited respectively.)

Listing 1 shows an attribution example (syntax slightly simplified for presentation purposes). Here, the attribute ↑`allServices` of a device is defined as the union of all its native services and all the synthesized services of its compositions. The compositions and services have attributes ↓`host` that refer to the hosting device. This attribute is defined by an equation on the `Device` node, which is an ancestor of compositions and services.




```
1  System ::= Device*;
2  Device ::= Native* Composition*;
3  abstract Service;
4  Native:Service;
5  Synthesized:Service;
6  Composition ::= Synthesized*;
7  rel Composition.connectedTo* <-> Service.connectedFrom*;
```

Figure 1: Basic model for running PALCOM system. *Upper:* conceptual model diagram. *Lower:* Corresponding abstract grammar.

Listing 1: Attributes defining derived properties

```
1  ↑ Device.allServices : 𝒫(Service)
2  ↓ Service.host : Device
3  ↓ Composition.host : Device
4  eq Device.allServices = Native ∪ (⋃_{c∈Composition} c.Synthesized)
5  eq Device.**.host = this
```

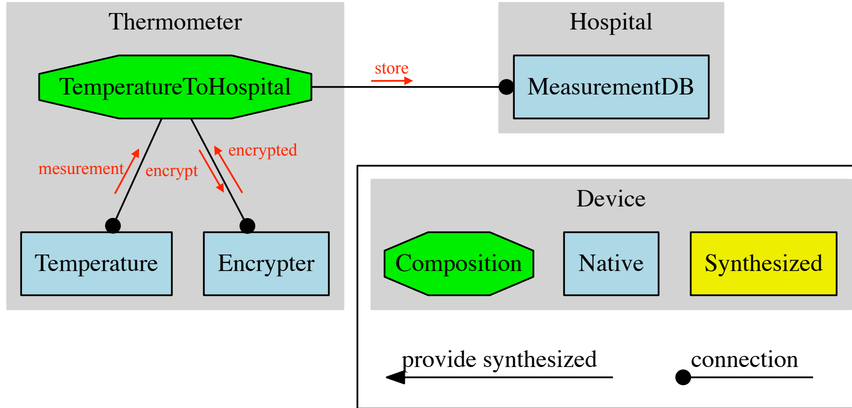


Figure 2: Overview of Mark's initial system. (See Figure 5 for example with synthesized service.)

3 Running example

As a running example of IoT runtime models and their analyses, we tell a story about the tech-savvy Mark and how he deploys an IoT system to communicate measurements to his physician[1].

3.1 A Simple IoT System

Mark has a chronic illness and needs to regularly inform his physician about his body temperature. The hospital provides a database service to which measurements can be sent from smart devices like thermometers, blood pressure monitors, etc. All the hospital staff have access to the database, but Mark would like only his physician to access his data, and would therefore like to encrypt his measurements with the public key of his physician.

Mark has a smart thermometer with a temperature service that sends a message whenever a measurement is taken. To build the system, Mark deploys an encryption service on the smart thermometer, and then creates a composition in COMPOS to connect the services. An overview of the system is shown in Figure 2.

The composition script is shown in Listing 2. It specifies what services to use and on what devices the services run (lines 2-4). Line 5 waits for the thermometer to send a measurement, storing it in the variable `t`. Lines 6-7 sends the temperature to the encryption service and receives the encrypted temperature. Line 8 sends the encrypted temperature to the measurement database at the hospital.

[1] We use the example as a pedagogical tool, and thus it is not entirely realistic.

Listing 2: Composition connecting services

```
1  composition: TemperatureToHospital
2   service tmp = Temperature on Thermometer
3   service enc = Encrypter on Thermometer
4   service mDB = MeasurementDB on Hospital
5   when receive measurement(var t) from tmp do
6    send encrypt(t) to enc
7    receive encrypted(var et) from enc
8    send store(et) to mDB
```

3.2 Updating the System

The system works fine, but Mark's illness makes it difficult for him to move around in the apartment. He therefore gets one more thermometer, so he can have one in his bedroom and one in the living room. He first updates the composition to work on any thermometer by changing line 2 to `service tmp = Temperature on this` (`this` binds to the device the composition is running on). He then copies the composition to the new thermometer, and it seems to work.

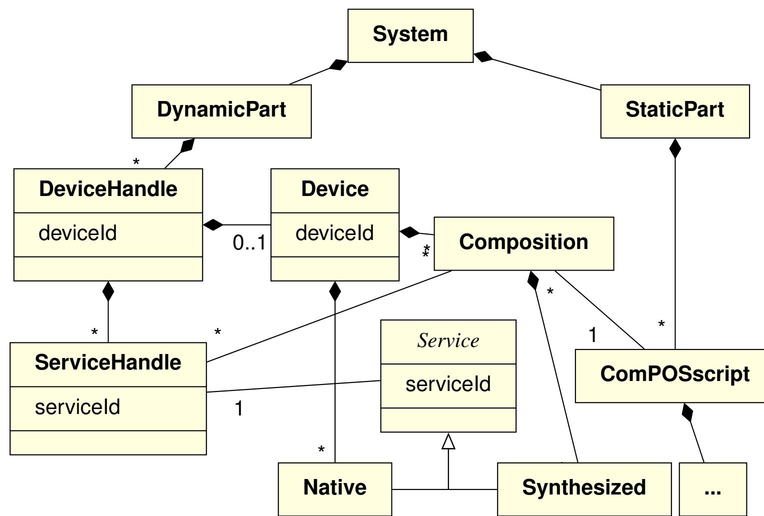
To get more confidence in that the system works as it should, he runs a *device dependency analysis* (DDA). With the DDA, he checks which devices are needed for his new thermometer to send a `store` message to the database. The DDA is run on the deployed system in order to resolve `this` expressions and to compute what concrete devices the system depends on. To Mark's surprise, the analysis reports that not only the new thermometer and the hospital database are needed, but the old thermometer is needed as well!

Mark looks at the composition again, and realizes that the old thermometer is used for the encryption. While this works, it means that the new thermometer will stop working if the old one runs out of battery. He fixes the problem by updating the compositions on line 3 to bind to the `Encrypter` on the same device (using `this` for the device).

4 The Full System Model

The full system model includes support for incomplete systems (due to unavailable devices) and composition scripts for allowing fine-grained analyses. Both the conceptual model and its corresponding specification using Relational RAGs is shown in Figure 3.

A `System` contains both a dynamic part for what is currently known about devices, service instances, and composition instances on the network, and a static part that contains the composition scripts. Each composition instance is related to its corresponding script (line 13 in the grammar). There is an abstract grammar for




```
1   System ::= DynamicPart StaticPart;
2   DynamicPart ::= DeviceHandle*;
3   DeviceHandle ::= <deviceId> ServiceHandle* [Device];
4   Device ::= <deviceId> Native* Composition*;
5   ServiceHandle ::= <serviceId>;
6   abstract Service ::= <serviceId>;
7   Native:Service;
8   Synthesized:Service;
9   Composition ::= Synthesized*;
10  rel Composition.connectedTo* <-> ServiceHandle.connectedFrom*;
11  rel ServiceHandle.service? <-> Service.serviceHandle;
12  StaticPart ::= ComPOSscript*;
13  rel Composition.implementation <-> ComPOSscript.instances*;
14
15  ComPOSscript ::= ...
```

Figure 3: Full system model with diagram and grammar (COMPOS details omitted)

the composition scripts as well, that we omit here for brevity, and that is reused from the implementation of COMPOS [Åke+19].

The dynamic part is similar to the basic runtime model in Figure 1, but introduces handles `DeviceHandle` and `ServiceHandle`. These are used for representing devices and service instances that a composition (tries to) connect to, regardless of if they are available on the network or not. The handles have optional relations to the corresponding true `Device` and `Service` entities, that are present in the model if they are available on the network.

4.1 Handling Unavailable Devices

Each device is identified by a globally unique id, *deviceId*, and each running service instance has a device-locally unique id, *serviceId*. A running service is globally identified with a tuple *(deviceId, serviceId)*.

The PALCOM middleware includes a discovery manager that can be run on any device. It keeps track of transitively available devices, as well as services and compositions that run on them.

To populate the model, the discovery manager is queried for its information, adding device handles, devices, service handles, services, and compositions. When a composition is added, it has information about each service instance it (tries to) connect to, i.e., the tuple *(deviceId, serviceId)*. If handles for the corresponding device and/or service are not already present in the model, they are added. At a later point in time, when new devices and service/composition instances are discovered, the model is automatically updated.

Figure 4 shows the object diagram from the perspective of Mark's home. If the hospital device is unreachable, the model will not contain the dashed elements. Devices, compositions, and services are shown in the same colors as in the overview figure.

5 Device Dependency Analysis

The Device Dependency Analysis (DDA) computes what sets of devices need to be available for a given message to be sent from a given composition and received by a given service.

5.1 Example

Consider the IoT system in Figure 5, where Mark has added two thermometers and moved the encrypter service to a router device. Because Thermometer2 uses Fahrenheit, Mark has created TempC, a synthesized service provided by TempFtoC, to convert the temperature to Celsius using a native service FtoC. The DDA could, for example, compute which devices are needed for the `store`

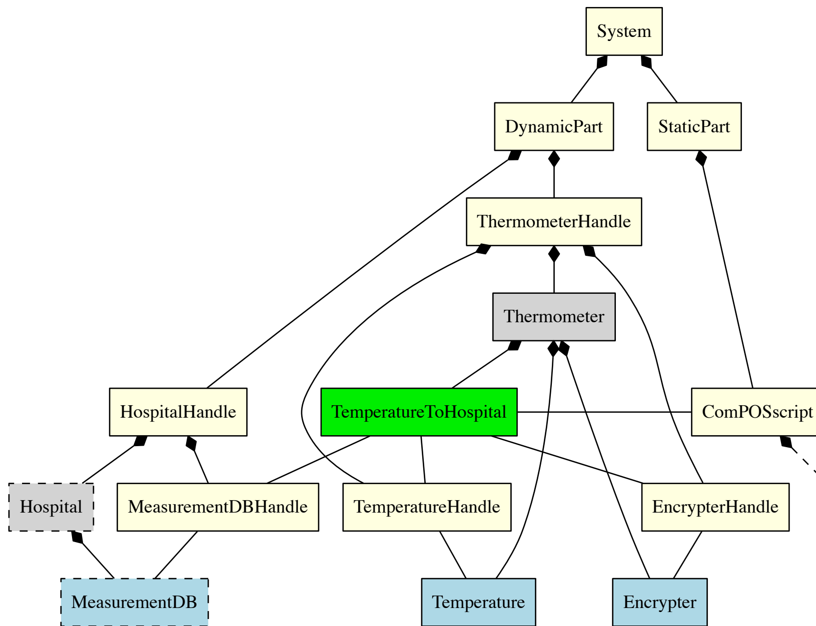


Figure 4: Object diagram over Mark's system

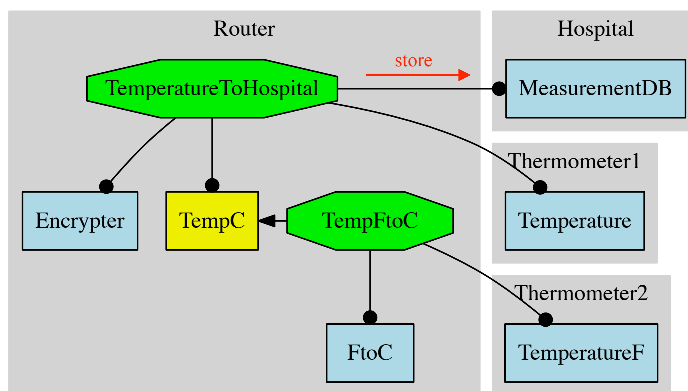


Figure 5: System with two thermometers and a router.

message to be sent. To compute this, the analysis takes into account the control flow for both *TemperatureToHospital* and *TempFtoC*.

The analysis takes three arguments: c, m, and s, where c is the sending composition, m is the name of the message, and s is the receiving service. The output of the analysis is a set of device sets $\{D_1, D_2, ...\}$, where each D_k is the set of devices needed in one control-flow that leads to c sending message m to s.

For the system in Figure 5, the DDA for the `store` message, i.e., `dda(temperatureToHospital, "store", MeasurementDB)`, would give the result $\{\ \{t_1, r, h\}, \{t_2, r, h\}\ \}$, where r is the router, t_1 and t_2 are the thermometers, and h is the hospital server. The result can be interpreted as a logical formula on disjunctive normal form, in this case $(t_1 \wedge r \wedge h) \vee (t2 \wedge r \wedge h)$, where a device is true if it is available on the network. The `store` message can be sent when this formula is true, i.e., when the router and hospital servers are available, and at least one of the thermometers.

5.2 Control-flow Analysis of Compositions

The control-flow analysis of the composition scripts is abstracted into two attributes $\uparrow$`ddaC` and $\uparrow$`ddaS`, that represent two kinds of control-flow. $\uparrow$`ddaC` computes dependencies needed for a composition to send a given message to a given service. $\uparrow$`ddaS` computes the dependencies needed for a synthesized service to send (or reply) a given message to some composition that connects to it. By introducing these attributes, the rest of the DDA can treat the actual composition language as a black box, see Listing 3. The equations for these attributes are omitted for brevity.

A dependency $dep \in Dep$ is relative to a given composition, and is modeled as a tuple `(serviceHandle, message)`, representing that a given service sends a given message to the composition. Both $\uparrow$`ddaC` and $\uparrow$`ddaS` return a set of sets of dependencies, i.e., a value of type $\mathcal{P}(\mathcal{P}(Dep))$, that we call a *dependency expression*, $\mathbb{E}$. It can be thought of as a logical expression in disjunctive normal form, similarly as for devices discussed earlier, but where a variable is a dependency.

5.3 Implementation of the DDA

We can now describe how the DDA is implemented for the conceptual model, using Relational RAGs, as shown in Listing 4.

To correctly analyze TemperatureToHospital and TempFtoC in Figure 5, the analysis needs to be transitive and follow dependencies between different compositions that are connected via synthesized services. In our analysis, we consider transitive dependencies only in the forward direction, when a composition sends a message through a synthesized service. To also consider messages received through a synthesized service, the analysis would need to be extended.

Listing 3: Control-flow of composition language

```
1  ↑ ComPOSscript.ddaC(c:Composition, m:String, s:Service) : 𝒫(𝒫(Dep))
2  ↑ ComPOSscript.ddaS(s:SynthesizedService, m:String) : 𝒫(𝒫(Dep))
3  eq ...
```

Listing 4: DDA analysis

```
1  ↑ System.dda(c:Composition, m:String, s:Service) : 𝒫(𝒫(D))
2  eq System.dda(c, m, s) = let 𝔼 = c.implementation.ddaC(c, m, s)
3   in let res = ⋃_{E∈𝔼} ⨂_{e∈E} e.serviceHandle.expand(e.message)
4     in toDevices(res) ⊗ { {c.host, s.host} }
5
6  ↑  ServiceHandle.expand(m): 𝒫(𝒫(Dep))
7  eq ServiceHandle.expand(m) =
8   this.hasService ? this.service.expand(m) : { {(this, m)} }
9
10 ↑  Service.expand(m): 𝒫(𝒫(Dep))
11 eq Native.expand(m) = { {(this.serviceHandle, m)} }
12 eq Synthesized.expand(m) =
13  let 𝔼 = this.composition.implementation.ddaS(this.serviceHandle, m)
14  in let res = ⋃_{E∈𝔼} ⨂_{e∈E} e.serviceHandle.expand(e.message)
15    in res ⊗ { {(this.serviceHandle, m)} }
```

The analysis makes use of the attributes ↑`ddaC` and ↑`ddaS` to compute control flow in composition scripts, and the attribute ↑`expand` to compute a transitive closure of all dependency expressions. It then projects the expanded dependency expression down to a set of sets of devices.

The calculation of the transitive closure is done by first calculating the `ddaC` (line 2) and then expanding each of the dependency sets in the resulting dependency expression. The expansion on line 3 uses the operator $\otimes : (\mathcal{P}(\mathcal{P}(Dep)), \mathcal{P}(\mathcal{P}(Dep))) \mapsto \mathcal{P}(\mathcal{P}(Dep))$, where $\mathbb{E}_l \otimes \mathbb{E}_r = \bigcup_{E_l \in \mathbb{E}_l, E_r \in \mathbb{E}_r} (\{E_l \cup E_r\})$.

For synthesized services, the expansion leads to calls to ↑`ddaS` for the composition containing the synthesized service (line 13). These dependency expressions are then expanded recursively (line 14), to compute the transitive closure.

For native services, the expansion cannot go further, and just returns the same expression that was expanded (line 11).

For an incomplete ServiceHandle (i.e., when there is no corresponding Service), the expansion also stops, and the same dependency expression is returned (line 8). This lack of information is also computed, so the user can see where the analysis is not complete, but the specification of this is left out for brevity.

The result of the DDA computation is shown on line 4: The fully expanded dependency expression is projected to the corresponding set of sets of devices, adding the devices of the original sending composition and receiving service to each device set.

5.4 Discussion

The DDA analysis is not trivial, and while it would have been possible to write it using an ordinary general-purpose language like Java, writing it using Relational RAGs gives a concise executable high-level specification. The performance of RAGs is on par with ordinary general-purpose languages, and a reason for this is that RAG evaluation is optimal in that each attribute value is computed at most once. This is achieved by automatic memoization of attributes during evaluation [Jou84].

6 Related Work

A recent survey [BGS19] of works in the area of models@run.time revealed two research challenges tackled by our approach. First, only few works directly address uncertainty, which in our work was taken into account using handles for both services and devices. Secondly, the need for distributed runtime models was identified, which we tackle by incorporating the PALCOM middleware.

A similar approach in the domain of smart homes was presented in [Sch+19]. There, the problem on how to connect multiple smart home middleware systems with different machine learning components was discussed and a runtime model to include all necessary information was presented as solution. Similar to our approach, RAGs were used to describe the model. However, the distributed nature of application in the domain of Internet of Things was not considered and left to the middleware systems.

Hartmann presented an approach in [Har16] for modeling large cyber-physical systems. Similar to our approach, the changing nature of runtime systems was in the focus, but they use streams to split up the complete model into atomic information and use analysis in both time and multiple worlds to support what-if-analysis. As an implementation basis, KMF [Fra+14] was chosen and extended to support a fixed set of derived properties, similar to attributes of RAGs.

Dai, Covvey, Alencar, and Cowan use logic programming to implement dependency analysis for workflows in [Dai+09]. Instead of having a model, they have facts, and instead of attributes, they have rules. The use of logical programming allows them to express different kinds of queries, which our current approach does not support. Compared to our work, Dai, Covvey, Alencar, and Cowan do not consider distributed systems and thus can do their analysis statically. They also explore data dependencies, which we might want to support in the future.

7 Conclusion

We have presented how Relational RAGs can be used for modeling and analyzing IoT systems with weak connectivity. The approach was evaluated by developing

a runtime model for the PALCOM IoT middleware, and implementing a forward transitive device dependency analysis as an example. The analysis can be implemented in a high-level compact way using the Relational RAGs.

In the future, we plan to use this work as a basis for exploring different kinds of analysis in IoT systems, such as information flow and placement of compositions, and thus helping developers find problems before making their systems available. It would also be interesting to implement the DDA in a conventional model transformation tool, for comparison.

8 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. It is also partly supported by the German Research Foundation (DFG) as part of Germany's Excellence Strategy – EXC 2050/1 "CeTI" and the project "HybridPPS" (project number 418727532), and using tax money based on the budget approved by the Saxon state parliament for the project "PROSPER".

References

[BGS19] Nelly Bencomo, Sebastian Götz, and Hui Song. "Models@run.time: a guided tour of the state of the art and research challenges". en. In: *Software & Systems Modeling* 18.5 (Jan. 2019), pp. 1619–1374.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B. France. "Models@run.time". English. In: *Computer* 42.10 (Oct. 2009), pp. 22–27.

[Dai+09] W. Dai, D. Covvey, P. Alencar, and D. Cowan. "Lightweight query-based analysis of workflow process dependencies". In: *Journal of Systems and Software* 82.6 (2009), pp. 915 –931.

[Fra+14] Fouquet Francois, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. "Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use". In: May 2014.

[Har16] Thomas Hartmann. "Enabling Model-Driven Live Analytics For Cyber-Physical Systems: The Case of Smart Grids". en. PhD Thesis. University of Luxembourg, Nov. 2016.

[Jou84] Martin Jourdan. "An Optimal-time Recursive Evaluator for Attribute Grammars". In: *International Symposium on Programming*. Vol. 167. Lecture Notes in Computer Science. Springer, 1984, pp. 167–178.

[Mey+20] Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. "Relational reference attribute grammars: Improving continuous model validation". In: *Journal of Computer Languages* 57 (2020). 100940.

[Sch+19] René Schöne, Johannes Mey, Boqi Ren, and Uwe Aßmann. "Bridging the Gap between Smart Home Platforms and Machine Learning using Relational Reference Attribute Grammars". In: *Proceedings of the 14th International Workshop on Models@run.time*. Munich, Sept. 2019, pp. 533–542.

[Sha93] Mary Shaw. "Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status". In: *WSSD 1993, Studies of Software Design*. Vol. 1078. LNCS. Springer. 1993, pp. 17–32.

[Ste+08] 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.

[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.

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, 2017, 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* [Ste+08]. 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

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

```
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

using aspects with inter-type declarations, like in MultiJava and AspectJ [Cli+00; 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.

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

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

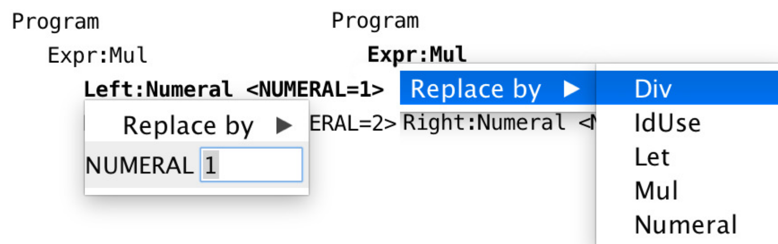


Figure 3: Default menus for `Mul` and `Numeral` nodes.

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.

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.

```
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.

4.2 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 `numeral`s can be edited or replaced using the menu on the `Mul` node.

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

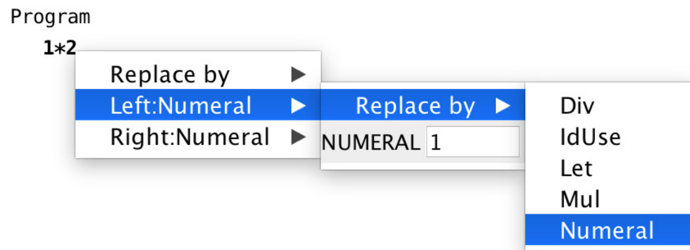


Figure 7: The menus for the hidden `Numeral` nodes have been merged into the `Mul` menu.

```
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`

`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.

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.

```
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

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, 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.

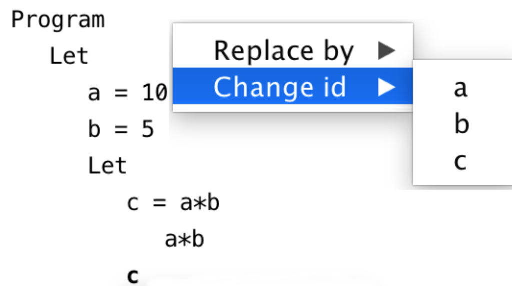


Figure 11: Intelligent code completion

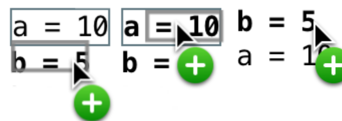


Figure 12: Using drag and drop to reorder a list

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.

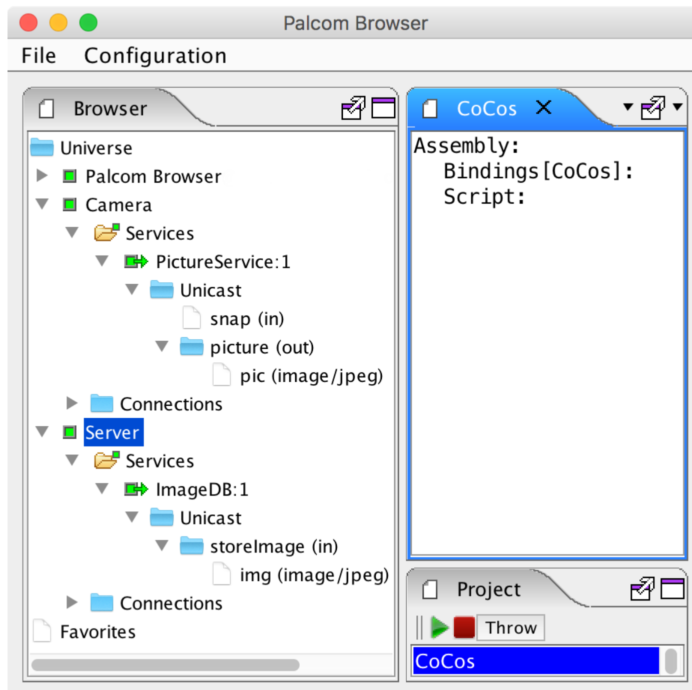


Figure 13: The PALCOM browser with a discovery panel to the left and an As2 script to the right

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.

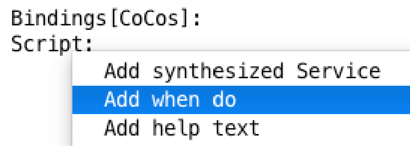


Figure 14: The menu for adding *when do*

```
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

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

```
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

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 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 [Ste+08], 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 [Kol+09], 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

[Cli+00] 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.

[Kol+09] 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] Steven 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.

[Ste+08] 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.