Developing Energy-Aware Software

Steven te Brinke

Developing Energy-Aware Software

Steven te Brinke

Composition of the Graduation Committee

Chairman: Prof. dr. Peter M.G. Apers
Promotor: Prof. dr. ir. Mehmet Akşit
Co-promotor: Dr.-Ing. Christoph Bockisch

Members:
Prof. dr. ir. Arend Rensink University of Twente
Dr. Luís Ferreira Pires University of Twente
Prof. dr. Uwe Assmann TU Dresden
Prof. dr. Vasily Moshnyaga Fukuoka University
Prof. dr. Wolfgang De Meuter Vrije Universiteit Brussel
Dr. Kerstin Eder University of Bristol

CTIT

CTIT Ph.D. Thesis Series No. 15-359

Centre for Telematics and Information Technology
University of Twente, The Netherlands
P.O. Box 217, 7500 AE Enschede

IPA Dissertation Series No. 2015-16

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Technology Foundation STW

The work in the thesis was supported by the ENOFES project (ENergy Optimization For Embedded Systems), funded by STW grant STW-11850.

ISBN 978-90-365-3875-6

ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 15-359)

DOI 10.3990/1.9789036538756

Available online at <http://dx.doi.org/10.3990/1.9789036538756>

Typeset with L^AT_EX

Cover design by Steven te Brinke

Image NGC6366 by Ron Brecher; Blue Marble, Earth at Night, and Andromeda by NASA

Printed by Ipskamp Drukkers, Enschede, The Netherlands

Copyright © 2015 by Steven te Brinke, Enschede, The Netherlands

DEVELOPING ENERGY-AWARE SOFTWARE

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 4 juni 2015 om 14:45 uur

door

Steven te Brinke
geboren op 29 juni 1986
te Doetinchem

Dit proefschrift is goedgekeurd door:

Prof. dr. ir. Mehmet Akşit (promotor)

Dr.-Ing. Christoph Bockisch (copromotor)

ACKNOWLEDGEMENTS

Attaining a doctorate degree consists of a lot of research and learning, but also includes many philosophical discussions along the way. Therefore, it is a path that you'd never walk alone. I'd like to thank anyone who accompanied me on the way, and will mention some of those who influenced me most.

First of all, I'd like to thank my daily supervisors, Lodewijk and Christoph. Lodewijk, even though you worked little time at the university, your enthusiasm for Co-op had considerable influence on my research. Christoph, I'm very glad that you could accompany me until the end, even though you moved to the Open University. You provided lots of valuable feedback and always managed to encourage me when times looked tough. I'd also like to thank my promotor Mehmet and colleague Somayeh, who provided valuable insights.

All members involved in the ENOFES project and its user board receive my appreciation for providing valuable feedback, and STW for funding my research. I'd like to thank my committee for reading my thesis and providing valuable feedback, which resulted in the booklet you're reading right now.

My research shifted along with the group, from Software Engineering in the TRESE group through ISSE to more formal research inspired by FMT. I'd especially want to thank FMT for the abundance of social events, first as "Floor 5" events, later being a proper member of FMT. In particular, I'd like to thank my colleagues Haihan, Brend, Lesley, and Buğra. As office mate, Haihan explained me the basics of Chinese culture and writing, though most is still Greek to me. Brend is a good friend since we started our studies together, and now he pursues a doctorate in the database group. Our contact intensified when I resided on the same floor as the database group and our discussions on how to organize data resulted in a joint publication.

Lesley encouraged both long scientific discussions and nice holidays. We won the CodeCup together and have discussed ages about database and programming language design. I hope I have positively influence his research with my discussions ;). Our holiday to Turkey, guided by Buğra, is also unforgettable. Buğra is always in for philosophical discussions. He tries to convert me to the dark side, but I'd love to remain on the light side.

Besides computer science, my love is speed skating, so I'd like to thank everyone from De Skeuvel who made sporting fun. In particular, my noabers Kim and Hans receive gratitude for all time spend together. And of course Michiel, Marijn, and Job, who taught me how to skate or how to teach skating. A lot of free time went into the Skeuvel website, a fun shared by Job, Jip, and Sybe. Two of these even became members of FMT :).

Besides regular sports, I pursued some other challenges. When winter was cold, I enjoyed time on natural ice with my family and best friend Willem. Even though Christoph might not fully understand this Dutch love, he did support it by flexibly changing schedules when winter became cold :). I cycled the Elfstedentocht with my uncles Henk and Louis and skated the alternative one with Henk. Hiking provided time for mentation and discussion, especially with Ronald, a good friend and former member of FMT, with whom I walked more miles than with anyone else. So special thanks go to these friends; with such people around, I'd certainly don't have to walk alone.

Last, but not least, I'd like to thank my dear family: my parents Mien and Leo, and my sister Linda with her boyfriend Rogier, for their unconditional love.

Steven te Brinke
Enschede, May 2015

CONTENTS

1	INTRODUCTION	1
1.1	Reducing Energy	1
1.2	Modularity of Optimizations	1
1.3	Programming Languages for Modularity	2
1.4	Models of Energy Behavior	2
1.5	How to Read This Thesis	3
i	PROGRAMMING LANGUAGES FOR COMPOSITION	5
2	CO-OP LANGUAGE DEFINITION	7
2.1	Flexible Semantics by Message Rewriting	7
2.2	Static View of Co-op	9
2.2.1	Classes	9
2.2.2	Method Definitions	10
2.2.3	Typing	11
2.3	Dynamic View of Co-op	11
2.4	The Co-op Language in Detail	15
2.4.1	Condition Language	15
2.4.2	Message Rewrite Language	17
2.4.3	Constraint Language	18
3	COMPOSING BEHAVIOR	23
3.1	Composition	23
3.1.1	Examples of Explicit Composition Mechanisms	24
3.1.2	Implicit Composition Mechanisms	24
3.2	Problem Analysis	25
3.3	Scope and Assumptions	26
3.4	General Approach	27
3.5	Co-op	29
4	COMPOSING DATA ACCESS	31
4.1	Keyword Free Composition Operators	31
4.2	Using a Composition Operator: Inheritance	32
4.3	Defining a Composition Operator: Inheritance	32
4.3.1	Defining Inheritance of Methods	33
4.3.2	Defining Inheritance of Fields	34
4.4	Providing a Composition Library: Inheritance	36
4.4.1	Inheritance Classification of Taivalsaari	36
4.4.2	Composition Library	39
4.5	Related Work	41
5	TAILORING CONTROL FLOW	45
5.1	Implementing Tailorable Control Flow	46

CONTENTS

5.1.1	Influencing Execution Order	46
5.1.2	Associating Scopes	46
5.2	Influencing Execution Order	47
5.2.1	Continuations	48
5.2.2	Exception Handling	49
5.2.3	Exception Handling with Continuations	49
5.2.4	Multi-Catch Exception Handling	52
5.2.5	Retrying and Resuming Exceptions	53
5.2.6	Additional Possibilities of Continuations	54
5.2.7	Weaknesses of Control-Flow Composition with Continuations	56
5.2.8	Conclusion	56
5.3	Associating Scopes	57
5.3.1	Scope Associations	57
5.3.2	Influencing Scopes and Execution Order	59
5.4	Related Work	59
ii	RESOURCE UTILIZATION MODELS	61
6	RUM NOTATION	63
6.1	Requirements for Resource-Aware Software	63
6.2	A Notation for Resource-Aware Software	64
6.3	Resource-Utilization Model (RUM)	65
6.4	Example Applications	66
6.4.1	The Hiker's Buddy Application	67
6.4.2	Smart Phone Network Traffic Reduction	69
6.4.3	Modularity of Case Studies	71
6.5	Formal Definition of RUMs	72
6.5.1	Identifying Key Properties	72
6.5.2	Modeling Resource Behavior	74
6.5.3	Analyzing Resource Behavior and Selecting Optimizer	75
7	DESIGN PROCESS	77
7.1	Overview	77
7.2	Design Activities in Detail	79
7.2.1	Analyze Resource Behavior	80
7.2.2	Model Resource Ports of Functional Components	80
7.2.3	Model Resource Behavior of Functional Components	81
7.2.4	Model User Components	81
7.2.5	Model Optimizer Components	82
7.3	Workflow using Uppaal	82
7.3.1	Overview of Uppaal	83
7.3.2	Design Activities	84
7.4	Media Player Example	85
7.5	Trade-offs	90
7.5.1	General Workflow	90
7.5.2	Uppaal-based Workflow	90

8	EXTRACTING RUMS	93
8.1	Tool Support	93
8.2	CEGAR	95
8.2.1	Example of Using CEGAR	96
8.3	Using MAGIC for Deriving RUMs	98
8.3.1	Automatically Simplifying Derived RUMs	99
8.3.2	Application of CEGAR	100
8.4	Model Checking RUMs	101
8.5	Related Work	101
8.6	Conclusion	102
9	DERIVING RUMS WITH PROFILING	105
9.1	Overview of CEGAR with Profiling	106
9.2	Details of CEGAR with Profiling	108
9.2.1	Abstracting	108
9.2.2	Model Checking	108
9.2.3	Profiling	109
9.2.4	Refining Abstraction	110
9.3	Tools for Profiling	111
9.3.1	Trepan	111
9.3.2	JouleUnit	111
9.3.3	SEFLab	111
9.4	Example	112
iii	BACK MATTER	117
10	CONCLUSION	119
10.1	Co-op	119
10.2	Resource-Utilization Models	120
A	CO-OP/II PROTOTYPE	121
A.1	Limitations	121
A.2	Performance	123
B	COMPOSITION LIBRARY	127
B.1	Details of Composition Operators	127
B.1.1	Static	127
B.1.2	Event Notification	128
B.1.3	Multiple Inheritance	129
B.2	Code of Composition Operators	134
B.2.1	Static Method Calls	134
B.2.2	Dispatch Optional	135
B.2.3	Method Inheritance	136
B.2.4	Field Inheritance	138
C	CO-OP/III	139
C.1	Implementation	140
C.1.1	Code Generation	140
C.1.2	CoopUnit	142

CONTENTS

C.1.3	Annotations	142
C.1.4	Java Integration	145
C.2	Limitations	147
C.2.1	Bindings and Constraints	147
C.2.2	Control-Flow Composition	147
C.2.3	Implicit Parameters	148
D	MEDIA PLAYER EXAMPLE	149
E	MAGIC RUM	153
E.1	Using MAGIC RUM	153
E.2	The Simplification Process	153
E.3	Predicate Simplification	154
E.4	Discussion	156
F	PROFILED CEGAR	157
G	PUBLICATIONS BY THE AUTHOR	161
H	SUPERVISED THESES	163
	BIBLIOGRAPHY	165

LIST OF FIGURES

Figure 1	Conceptual message send	7
Figure 2	Actual message send	8
Figure 3	Duplication during message send	8
Figure 4	Multiple stages during message send	8
Figure 5	Example of the <i>skip</i> (defaultBinding, virtualBinding) constraint	20
Figure 6	Working of pre constraint	21
Figure 7	Working of cond constraint	21
Figure 8	Working of skip constraint	21
Figure 9	Example of the composition operator for inheritance	27
Figure 10	Composition operators reroute messages	28
Figure 11	Overview: Composition in <i>Co-op</i>	29
Figure 12	Composition in <i>Co-op</i> : Delegation	30
Figure 13	Basic inheritance DAG	37
Figure 14	Both directions of lookup	38
Figure 15	Vertical overlapping	38
Figure 16	Horizontal overlapping	38
Figure 17	Example of calcc	48
Figure 18	Using try catch finally with continuations	51
Figure 19	Notation for resource-aware components	65
Figure 20	Energy state chart	66
Figure 21	A component diagram for Hiker's Buddy	68
Figure 22	A component diagram for smartphone network performance	70
Figure 23	Example over and under-abstraction for a coffee machine	74
Figure 24	Relation between actual system and its abstractions	75
Figure 25	Workflow for designing resource-aware software	78
Figure 26	Simple UPPAAL timed automaton for a coffee machine	84
Figure 27	Media player high-level diagram	86
Figure 28	Media player detailed diagram	87
Figure 29	Media player overview	89
Figure 30	User behavior (partial scenario)	89
Figure 31	Analysis of scenario	90
Figure 32	Design method for resource-aware software	94
Figure 33	Steps of CEGAR	95
Figure 34	Simple over-abstraction	97
Figure 35	Slightly more detailed abstraction	98
Figure 36	State of refined model generated by MAGIC	99
Figure 37	Excerpt of simplified over-abstraction	100
Figure 38	Steps of CEGAR	105
Figure 39	Example behavior	106

Figure 40	Actual energy consumption	106
Figure 41	Example energy profiles	107
Figure 42	Precision of energy profiles	107
Figure 43	Jbcpp model of source code (to illustrate the size)	113
Figure 44	Jbcpp model after grouping	114
Figure 45	UPPAAL model	115
Figure 46	Trepan profile	116
Figure 47	Debugging of a function call using inheritance	122
Figure 48	Class diagram of dispatch shown in figure 47	123
Figure 49	Difficult function call using multiple inheritance	125
Figure 50	Root of the dispatch tree shown in figure 49	126
Figure 51	Static dispatch example for the class shown in appendix B.2.1	128
Figure 52	Construction of an instance of SomeClass	129
Figure 53	Dispatch when a call is annotated with @ImplementationOptional	130
Figure 54	Staff member class diagram	130
Figure 55	Class and object diagram of a professor	133
Figure 56	Example of testing Co-op/III with JUnit	143
Figure 57	Java integration	146
Figure 58	Various controllers	150
Figure 59	Analysis of scenario with various network characteristics	151

LIST OF TABLES

Table 1	Example message properties	12
Table 2	Common message properties	13
Table 3	Example of ternary operators	16
Table 4	Binary selector operators	17
Table 5	Annotation operators in action selectors	18
Table 6	Constraints	20
Table 7	Delegation versus concatenation	37
Table 8	Variations of vertical lookup combination	38
Table 9	Supported inheritance properties by various languages	40
Table 10	Parameters of the network	149

LIST OF LISTINGS

Listing 1	Simple Co-op class defining a string prefixer	9
Listing 2	Traditional try-finally	46
Listing 3	First-class tryFinally	46
Listing 4	Java-like try-with-resources	47
Listing 5	First-class try-with-resources	47
Listing 6	Using try catch finally in Scala	49
Listing 7	Using try catch finally with continuations	49
Listing 8	Defining try catch finally with continuations	50
Listing 9	Exceptions with multiple catch blocks	52
Listing 10	Defining try catch finally, allowing multiple catch blocks	53
Listing 11	Catch block using retrying exceptions	54
Listing 12	Catch block using resuming exceptions	54
Listing 13	Defining try catch finally, allowing catch blocks to retry	55
Listing 14	Default scoping	57
Listing 15	Desired scoping	57
Listing 16	Definition of tryFinally	58
Listing 17	Definition of tryWithResources	58
Listing 18	Dispatch for static method calls	134
Listing 19	Dispatch optional	135
Listing 20	Method inheritance	136
Listing 21	Field inheritance	138
Listing 22	Hello world in Co-op/III	142
Listing 23	Unit test in Co-op/III	144
Listing 24	Cyclic annotation A	145
Listing 25	magic_rum-NetworkManager_key3.py	154
Listing 26	The script magic_rum.py	155
Listing 27	Example Java class on which we apply CEGAR	158

1 | INTRODUCTION

1.1 REDUCING ENERGY

The rise of resource-constrained devices, such as cell phones, but also the increasing awareness of the need for environmental sustainability, makes optimization of energy consumption an evermore important requirement [37]. Energy is in fact one of many resources that may need to be managed by software, and reducing energy consumption cannot be considered without taking into account the trade-offs with other resources (e.g., memory and bandwidth usage) and services (e.g., delivered quality of audiovisual artefacts).

Software systems are composed of various kinds of components organized in multiple layers. Typical examples are hardware-related software components, middleware components and application components. It has been demonstrated repeatedly that optimization techniques, implemented in software, can lead to substantial reduction of resource consumption [34, 81], within both the computer system and the system being controlled. Optimization can take place for components residing at each layer [34, 88, 91] and can be performed across layers. Our focus is on the reduction of energy consumption by controlling various external hardware components that are energy intensive. Such hardware components are typically represented by dedicated software components, such as device drivers, and can in this way be controlled by software.

1.2 MODULARITY OF OPTIMIZATIONS

The functionality of energy optimization can be embedded in the functional components of software. However, today's software is already facing the problem of complexity [86], and extending its functionality with energy optimizers increases this problem further. For example, if a functional component evolves such that functionalities are added or removed, its energy consumption may change, and the energy optimizers must be changed accordingly to consider the new energy behavior of the component. Embedding optimizations into functional components also causes tangling of optimization and functional behaviors, which increases the complexity and decreases the understandability of the software.

In the software engineering literature, modularization is commonly considered as a means to cope with the complexity of the software, because the scope of focus can be reduced to individual modules [74]. In the literature, a module is defined as a reusable software unit with well-defined interfaces and an implementation. Modules communicate with one another through their interfaces, which implies that the interfaces must convey sufficient information to facilitate the communication.

To cope with the complexity of energy-aware software, we claim that energy optimizers must be allocated to separate modules than the rest of software. This can easily be understood as the separation of the functional concerns and optimization concerns. To this aim, functional and optimizer components must provide suitable interfaces to each other, so that necessary information for performing the optimization can be exchanged among them.

1.3 PROGRAMMING LANGUAGES FOR MODULARITY

Energy is consumed during the actual execution of the software. Therefore, it is inevitable to carry out optimizations at runtime, which leads to self-adaptive software that adjusts its behavior based on changes in resource availability [21]. In order to support such software design, it must be possible to use energy models at runtime without compromising the modularity of the implementation. Even in a flexible programming language such as Python, it is not possible to implement energy behavior in a fully modular way [T.7]. Therefore, the first part of this thesis introduces a programming language, named Co-op, that aims to address this issue. Co-op can be positioned between AOP and a MOP: it is less expressive than a full MOP, but more expressive than AOP, because as well too expressive as too limiting languages reduce the readability of the source code.

Chapter 2 provides the definition of the Co-op language, after which we show how the language supports software composition. Co-op supports influencing the composition of behavior (chapter 3), data access (chapter 4), and advanced control flow (chapter 5).

1.4 MODELS OF ENERGY BEHAVIOR

Resource optimization can also be carried out statically before the execution of software, on which we will focus in the second part of this thesis. Static optimization is usually achieved by modeling the resource consumption at the architectural level and performing analysis on the models. Such models are needed to understand the energy behavior of the system and to analyze which control strategy reduces the energy consumption. Thus, the energy consumption must be specified inside these models, but existing modeling languages do not offer generic support for specifying energy behavior. Therefore, we introduce a modeling notation that supports energy behavior in chapter 6.

Designing models is not trivial either, so we also present guidelines for designing models that specify energy behavior in chapter 7. Finally, we conclude with a method to automatically extract models, including energy behavior, from source code (chapter 8) together with profiling of the application (chapter 9).

1.5 HOW TO READ THIS THESIS

This thesis is organized in two parts. The first part explains our programming language for modularity, the second part focuses on modeling resource consumption. Details about our implementations that are omitted in the main text are provided in the appendices. Also, the publications by the author and the theses co-supervised by the author are listed in the appendices. When cited, these publication can be distinguished respectively by the P and T that are prefixed before the number. The publications listed in the bibliography are numbered without letter.

Part I

PROGRAMMING LANGUAGES FOR COMPOSITION

2

CO-OP LANGUAGE DEFINITION

This chapter explains the semantics and syntax of the Co-op language, based on my master thesis “First-order function dispatch in a Java-like programming language” [P.1]. The core object language and basic syntax of Co-op are inspired by the Java programming language. However, we believe that many concepts of Co-op can be written using a syntax based on any object oriented language. Thus, we consider the concrete syntax of lesser importance than the semantics. Therefore, we will not elaborate all syntactic details and focus on the parts that are important for understanding the semantics.

2.1 FLEXIBLE SEMANTICS BY MESSAGE REWRITING

We observed that modularity constructs in programming languages are essentially late binding of functionality (such as a method implementation) to the use of a symbol (such as calling the method by name), together with more or less complex means to influence this binding. Co-op aims to provide a uniform and highly expressive way to define all kinds of modularity constructs in terms of message rewriting.

In Co-op, every method call and field access is dispatched using message sends. This section will present the basic idea of message sends using a metaphor. The details of dispatch will be explained in section 2.3.

Sending a message to a friend can be as easy as writing his address on a postcard and putting it in the postbox. Then, we assume it will be delivered to the intended receiver as shown in figure 1. In general, we do not think about the intermediate steps of the message delivery. As figure 2 shows, the message will be handled by a postal service, which is a black box to us. The postal service can influence the delivery in several ways, of which we will give a few examples:

- When the intended receiver is on holiday, the postal service might reroute the message to his holiday address.
- The message might be handled by another person than the intended receiver, for example his secretary.

Figure 1: Conceptual message send

Figure 2: Actual message send

Figure 3: Duplication during message send

- A security check can be done by Customs when the message crosses a country border. In such a case, the contents of the message are inspected and might be altered.

In any of these situations, there is a stakeholder that benefits from the ability to influence the message delivery process. In general, a consumer can easily subscribe to and unsubscribe from the services offered to consumers, such as the rerouting to a holiday address, without thinking of the influence these subscriptions have on the process of the postal service.

In Co-op, message sends are handled similar to the process described above. The message delivery process—postal service—is called *dispatch* and can be influenced by programmers. A single influence—consumer service—is a *composition operator*. A composition operator can be activated and deactivated by an application programmer—which is similar to subscription by consumers. Ideally, an application programmer does not have to think about the dispatch once he has activated the right composition operators, in the same way we do not think about the process of the postal service.

Co-op does not have a clear distinction between the address and contents of a message—a message is more like a postcard than an enveloped letter. Any composition operator has access to the full content of the message. Thus, dispatch can influence the full message, like Customs can influence your full package.

During dispatch, a message can be copied, as shown in figure 3. It is also possible that the message goes through multiple stages of dispatch, shown in figure 4. When we look at a specific dispatch, we say that the incoming message is the *original message* and the outgoing message is the *rewritten message*. The *initial message* is the message originally sent. Further details of dispatch will be explained in section 2.3.

Figure 4: Multiple stages during message send

```

1 class Prefixer {
2     var prefix;
3
4     method setPrefix(prefix) {
5         this.prefix = prefix;
6     }
7
8     method prefix(str) {
9         return this.prefix + str;
10    }
11 }

```

Listing 1: Simple Co-op class defining a string prefixer

2.2 STATIC VIEW OF CO-OP

2.2.1 Classes

Co-op is a class-based language [103]. Thus, Co-op does not provide built-in inheritance, a concept used by object-oriented languages. Inheritance can be achieved by providing the desired dispatch, as will be explained in chapter 4.

Co-op classes do not have a special constructor, but every class object has a method which returns a new instance of its type. An instance of `SomeClass` can be created as follows:

```
SomeClass.new();
```

An example Co-op class shown in listing 1. This class specifies a simple string prefixer. The class can be used as follows:

```

1 var prefixer = Prefixer.new();
2 prefixer.setPrefix("[mesg] ");
3 System.println(prefixer.prefix("Hello World!"));

```

Executing this example will result in the following output:

```
[mesg] Hello World!
```

Classes are loaded lazily in Co-op, like it is done in Java. When a class is loaded, the method `initializeClass()` will be called. This method fulfils the role of a static initializer. In Co-op, everything is an object. Thus, Co-op does not provide separate primitive types. The keyword `null` references the singleton instance of the class `coop.lang.Null`.

The structure of a Co-op class is similar to that of a Java class. Besides variables and methods, which are well-known members in object-oriented languages, Co-op also provides a declarative syntax to specify dispatch. Dispatch declarations never specify the full dispatch process, rather are always partial—like consumer services never influence the full process of a postal service.

Examples of more advanced classes utilizing dispatch are shown in the next chapters. A Co-op class can contain the following members:

- Variables: define named storage locations, as is common in imperative programming languages.
- Methods: define procedures inside classes.
- Conditions: define Boolean conditions over messages.
- Bindings: define which message is bound to which dispatch.
- Constraints: define dependencies and ordering among bindings.

VARIABLE A variable declaration defines the existence of a certain field inside the class. For example, the field `prefix` is defined on line 2 of listing 1.

METHOD A method definition defines a method signature and its body. Two methods with a single argument are specified in listing 1 (on lines 4 and 8).

BINDING A binding is structured as follows:

```

1 BindingName = (MessageSelector)
2 {
3     // Message rewrite
4 };

```

The *BindingName* is the name of the defined binding. The *MessageSelector* selects which messages are influenced by this binding—for example only messages to a certain person. The message rewrite is used to change values of the message—for example rewriting the address to the holiday address of the selected person.

CONSTRAINT Constraints express ordering and dependencies between bindings. For example, `pre(myBinding, defaultBinding)` specifies that `myBinding` should be executed before `defaultBinding`.

2.2.2 Method Definitions

Co-op methods are identified by their name and number of explicit parameters. Besides these properties, every method utilizes a—possibly empty—set of implicit parameters. Implicit parameters are parameters which are implicitly passed to a method. A method definition must define all used implicit parameters explicitly.

A message property is passed to the called method implicitly, if the method declares the use of this property as implicit parameter. By default, every method has the implicit parameter `this`. Which implicit parameters are used by a method can be defined by using the annotation `@ImplicitParameters` as follows:

```

1 method @ImplicitParameters(["this", "thisJoinPoint"]) someMethod() {
2     // Method body
3 }

```

Here, the method `someMethod` has two implicit parameters: `this` and `thisJoinPoint`. These two parameters can be used in the same way and have the same scope as explicit parameters. However, at the call side, these parameters are not explicitly passed to the method, shown in the following call:

```
someObject.someMethod();
```

From the perspective of the method body, the following method definition is equal to the previous one, even though the signatures differ:

```
1  method @ImplicitParameters([]) someMethod(this, thisJoinPoint) {
2      // Method body
3  }
```

At the call side, this method differs from the previous one, because we have to pass the parameters explicitly now:

```
SomeClass.someMethod(someObject, someJoinPoint);
```

Thus, we see that implicit parameters are implicit only in the way they are passed to a method. A method definition must define all used implicit parameters explicitly. The execution of a method is only possible if all implicit parameters are available, as will be explained in detail in section 2.3.

The body of a method consists of the same expressions allowed by Java. The only semantic difference is that field accesses and method calls are message sends. Besides annotating elements which allow annotations in Java, it is also possible to annotate message sends. Details about message sends are explained in section 2.3.

2.2.3 Typing

To allow the flexibility to deploy new composition operators at runtime, Co-op is typed dynamically. Composition operators are applied dynamically and can change the behavior of a class. For example, a composition operator might change the available methods on an object. Thus, at compile time it is not always known which methods will be available during runtime, making static type checking impractical; either it cannot guarantee safety, or it will be too restrictive by disallowing uncertain cases that are actually correct.

2.3 DYNAMIC VIEW OF CO-OP

In Co-op, both method calls and field accesses are modelled as message sends. The dot is the message send operator, so message sends can be written in the following way:

```

      1
    {
this.someOtherField = this.someField;
    }
      2
    {
this.someFunction();
    }
      3
```

Property	Message send 1	Message send 2	Message send 3
messageKind	"Lookup"	"Lookup"	"Call"
name	"someField"	"someOtherField"	"someFunction"
parameters	[]	[this.someField]	[]
target	this	this	this

Table 1: Example message properties

This code contains three message sends, namely:

1. A message of the kind “Lookup” with name “someField” and no parameters, which commonly results in a field read of someField.
2. A message of the kind “Lookup” with name “someOtherField” and a single parameter, namely the value returned by message send 1, which commonly results in storing this value in field someOtherField.
3. A message of the kind “Call” with name “someFunction” and no parameters, which commonly results in the execution of the method someFunction.

As can be seen in the three message sends above, every message send defines certain properties of the message (summarized in table 1). Besides the properties explicitly given in these three examples, a few other properties are also defined. For example, in all three cases the target of the message is *this*. Also, there is a sender property which is initialized to *this*. The common properties of a message are listed in table 2. The default properties are initialized by any message send, the other properties might be undefined, depending on the context. For example, when a message is sent from a static context—a context without a *this* reference—the sender of the message will be undefined.

An interesting message property is the set of call annotations. Annotations provide the ability to add additional information to a message send. Annotations can be added to every message send at the call side as follows:

```
this.@SomeAnnotation @OtherAnnotation("SomeValue") someFunction();
```

Besides the properties listed in table 2, a message can have any other property, because the availability of properties can be influenced by message rewrites, which will be explained in section 2.4.2. The default dispatch, explained in the next paragraph, uses certain properties of the message, as shown by the *use* column of table 2.

DEFAULT DISPATCH In order to achieve the common behavior of message sends described at the beginning of this section, Co-op provides a default binding which achieves this behavior. The default binding is always present, but only succeeds if the message addresses a declared method or field.

A message addresses a method if and only if (a) the messageKind is “Call”, (b) the targetType has a method named the value of name, (c) this method has the same amount of parameters as the length of parameters, and (d) for this method all implicit parameters are defined. Implicit parameters are named parameters which are automatically passed to the

Name	Def.	Type	Use	Meaning (of initial value)
sender	no	<i>any</i>		Originating context (null if static)
senderType	yes	Class		Type of originator
target	no	<i>any</i>	F	Intended receiver (null if static)
targetType	yes	Class	M	Type for method lookup
name	yes	String	MF	Name of method to be invoked
parameters	yes	list of <i>any</i>	MF	Explicit parameters to be passed to the method
messageKind	yes	"Call" or "Lookup"	MF	Whether the message should result in a method call or a field access
annotations		set of Annotation		Annotations of the call
this	no	<i>any</i>		Original target (not used for selection, but used as implicit parameter by many methods)
thisType	yes	Class		Class of the original target (not used for selection, but used as implicit parameter by methods)
result	no	<i>any</i>		Return value of last invoked method
error	no	<i>any</i>		Error thrown by last invoked method

Table 2: Common message properties

method when defined as message properties. For example, most methods use the implicit parameter `this`, but it is possible to require any number of implicit parameters.

A message addresses a field if and only if (a) the `messageKind` is “Lookup”, (b) the `targetType` has a field named the value of `name`, and (c) the length of `parameters` is (i) zero—field get—or (ii) one—field set. In the former case, the current value of the field will be returned. In the latter case, the value of the given parameter will be stored in the field and this value will be returned.

An overview of the properties used for addressing methods and fields is shown in table 2. The *use* column shows what the default dispatch uses this property for. This is for locating a method implementation (M) or a field instance (F).

We defined the default binding to be always present, but only sometimes successful. This is similar to saying a method call succeeds if and only if the called method is implemented. It is simple to imagine the default binding conforming to this definition:

```

1  binding defaultBinding = (true)
2  {
3      // The actual dispatch is performed: executing the method or accessing the field
4      // Only successful if the dispatch succeeds (i.e. method or field is defined)
5  };

```

BINDINGS Every binding consists of a condition, termed *message selector*, and an assignment block, termed *message rewrite*. The message selector specifies for which messages the binding is applicable. The message rewrite creates a modified copy of the message, which

is then resent. This rewritten message will be processed by all applicable bindings, exactly like the original message send. A binding is successful if and only if the message send of the rewritten message is successful.

For multi-level dispatch in general and single inheritance in particular, we can easily see that this recursive definition of success is desired. When modeling single inheritance, messages are resent to the super class recursively, as long as no implementation is present. Once an implementation has been found, the call succeeds. Thus, the call succeeds if and only if at least one super class implements the method. We will see later that more difficult situations, such as Beta inheritance, can be modeled using this recursive success as well.

Every binding *b* is a member of a class *C* and can access any property *p* of class *C*. This general structure is shown in the listing below. Any instance *c* of *C* has also an instance *c.b* of binding *C.b*.

```

1  class C {
2      var p;
3      binding b = (this.p) { ... };
4  }
5
6  var c = C.new();

```

We see here that we can distinguish between a binding class, *C.b*, and a binding instance, *c.b*. We talk about bindings when the distinction between binding instances and binding classes is not important.

A composition operator, which represents a concern, should be modeled by a set of binding classes. The instances of these binding classes represent the use of this operator. For example, inheritance is modeled by several binding classes. So for every inheritance relation, instances of these binding classes are created.

BINDING ACTIVATION In order to use a binding, it must be activated first. The activation of a binding *c.b* can be done by calling *c.b().activate()*. When the binding is no longer needed, it can be deactivated by calling *c.b().deactivate()*.

MESSAGE SENDS A message send succeeds if and only if at least one binding processes the message successfully. Whereas message resends are not required to succeed, in principle message sends written by a programmer are. In general, whenever a programmer writes a method call, he expects something to happen. Thus, the call should at least be dispatched to some implementation. If all bindings fail, the call cannot be dispatched to anywhere, so an exception is thrown.

However, if the programmer would like to signal an event which does not have to be handled, he can do so by adding an annotation to the call. For example, the built-in annotation `@ImplementationOptional` allows message sends which are not handled by anyone. The working of this annotation is defined in Co-op itself, it is not part of the language definition (appendix [B.2.2](#) shows the details).

2.4 THE CO-OP LANGUAGE IN DETAIL

2.4.1 Condition Language

A condition is essentially a Boolean expression over properties of the message. It can also execute message sends, for example to read fields of the object it is part of. Because in theory all message selectors, which are conditions, will be evaluated on every message send, it is important to allow optimization of these conditions. Therefore, conditions should be free of side effects. When all conditions are side-effect free, the runtime environment can—without influencing the result of the program—omit evaluation of any condition of which it has already computed the result. Because message sends are allowed in conditions, Co-op cannot guarantee that conditions are side-effect free. This is the responsibility of the programmer. Co-op only guarantees correct behavior if all conditions are side-effect free.

When the evaluation of expressions is performed lazily from left to right, which is the case in many programming languages, it is up to the programmer to write the cheap comparisons first. However, in conditions it is quite difficult to identify which operators are cheapest, because every condition only specifies a small part of the dispatch. The programmer is unaware of the total dispatch. Therefore, the programmer should not try to optimize the total dispatch, the execution environment should perform such optimizations. An example of such an optimization could be utilizing a binary decision diagram of all activated selectors, which allows evaluating as few conditions as possible.

In order to optimize, the execution environment should be able to reorder the evaluation of primitive expressions which are part of conditions. Therefore, the programmer should not expect a specific evaluation order based on the order in which expressions are defined. To distinguish between the semantics of common Boolean expression—which guarantee an evaluation order—and unordered behavior, we use a different syntax for the Boolean *and* (&) and *or* (|) operators inside conditions. These operators do not guarantee the order of evaluation, or that the evaluation is performed lazily. Because all operands are side-effect free, laziness does not influence the result, so it is an optional optimization strategy.

When unordered evaluation is used, it is desired that expressions possess the Church-Rosser property [22, 102]: “If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders, then all of these evaluation orders yield the same result.”

Boolean logic is not sufficient to guarantee this property for unordered evaluation, because dispatch can fail and we do not require that all expressions are evaluated. Thus, some evaluation orders might fail, whereas others succeed, of which an example is given in the next paragraph. To ensure that all evaluation orders yield the same result, Co-op uses ternary logic [60] for evaluating conditions. This means that we introduce an additional truth value: *unknown*. We can think of *unknown* as a sealed box containing either unambiguously true or unambiguously false. Some logical operators can yield an unambiguous result, even if they involve an *unknown* operand, as shown in table 3.

An example of a condition requiring ternary logic when evaluation is unordered, is the following condition:

```
a != null & a.length == 10
```

a	b	$a \wedge b$	$a \vee b$
true	true	true	true
true	false	false	true
true	unknown	unknown	true
false	true	false	true
false	false	false	false
false	unknown	false	unknown
unknown	true	unknown	true
unknown	false	false	unknown
unknown	unknown	unknown	unknown

Table 3: Example of ternary operators

When *a* equals null, it is clear that the condition should evaluate to false and that dispatching *a.length* would fail. Using left to right short-circuit evaluation results in $false \wedge \dots = false$. However, we do not require left to right evaluation, so the given condition is equal to:

```
a.length == 10 & a != null
```

When evaluated in this order, *a.length* will be evaluated first, which fails. Thus, without the use of ternary logic, unordered short-circuit evaluation can yield different results for different permissible execution orders. Therefore, we define the return value of a failing dispatch to be *unknown*. Now, evaluation will result in:

$$\begin{aligned}
 & unknown \equiv 10 \quad \wedge \quad a \neq null \\
 = & \quad unknown \quad \wedge \quad a \neq null \\
 = & \quad unknown \quad \wedge \quad false \\
 = & \quad false.
 \end{aligned}$$

When looking at the unordered *or* operator, we see that the result of both orders is also the same:

```
a == null | a.length == 10
```

Using left to right short-circuit evaluation when *a* equals null results in $true \vee \dots = true$. The given condition is equal to:

```
a.length == 10 | a == null
```

Evaluation of this order will result in:

$$\begin{aligned}
 & unknown \equiv 10 \quad \vee \quad a \equiv null \\
 = & \quad unknown \quad \vee \quad a \equiv null \\
 = & \quad unknown \quad \vee \quad true \\
 = & \quad true.
 \end{aligned}$$

Besides the operators explained above, a message selector can use normal binary operators and a special annotation operator. The possible operations are listed in table 4. The annotation operator is provided because annotations are an important property of messages,

matching type	operators	lhs	rhs
lazy	$\&$, $ $	object	object
normal	$==$, $!=$, $<$, $>$, $<=$, $>=$	object	object
annotation presence	$@==$, $@!=$	message	annotation matching expression

Table 4: Binary selector operators

adding annotations is the way for programmers to add additional information to a message send. In order to check for presence or absence of some annotation, respectively use one of the following expressions:

```
1 message @== @SomeAnnotation
2 message @!= @SomeAnnotation
```

Checking that the annotation has certain parameter values is also possible:

```
1 message @== @SomeAnnotation(name == "SomeName")
2 message @== @SomeAnnotation(priority > 4)
```

Recursion Avoidance

Infinite recursion is easily generated by message selectors. For example, by the following condition, which is used for defining inheritance:

```
messageKind == "Lookup" & target == this.child
```

This condition will generate a message send for reading the field `child`. In order to process that message, all message selectors, including the one just given, should be evaluated. This will result in the same message send again, causing infinite recursion.

Because message selectors are side-effect free, the evaluation of a message selectors does not change the state of the system. The behavior of a message send is only influenced by the state of the system and the message itself. When message selectors generate messages recursively, the state remains the same. Thus, if the message being processed is generated again recursively, it will always result in infinite recursion, which is never desired.

To avoid such infinite recursion, we define that whenever during the evaluation of a message selector, a new message is generated which is exactly the same as any of the messages causing the evaluation of this selector, the result of this selector is *unknown*. This avoids the most common infinite recursion.

In the example given at the beginning of this paragraph, now the given selector will be canceled for the message send for reading the field `child`, so the field read can happen normally. See Te Brinke [P.1] for more details on recursion avoidance.

2.4.2 Message Rewrite Language

A message rewrite creates a copy of the original message, modifies it and resends this modified message. The modification of the copy can be done using assignments:

Operator	Action
@+=	Adds the given annotation to the annotations of the message.
@-=	Removes the given annotation from the annotations of the message.

Table 5: Annotation operators in action selectors

```
mySelector = message.name;
```

The left hand side of the assignment references a property of the new message which should be set. The right hand side is an expression, in this case a message send for reading the field name of the original message.

For annotations, special operators are provided, which can be used as follows:

```
message @+= @MyAnnotation;
```

The left hand side is always message, as we are changing the annotations of the copy of the message. The right hand side is an expression which returns an annotation. In this example, we simply used an annotation literal. The possible operators are shown in table 5. Besides assigning values to properties, it is also possible to remove a property from the message:

```
remove mySelector;
```

None of these operations influence the original message. Thus, even after removing a property from the rewritten message, it will still be available on the original message.

2.4.3 Constraint Language

Multiple composition operators can apply at the same time. In order to express relations among these operators, we use constraints. Co-op models composition operators as binding classes. Therefore, constraints are expressed over binding classes.

A constraint specifies a relation between two binding classes. For example, $pre(A, B)$ specifies a precedence relation between binding A and binding B. The meaning of all constraints provided by Co-op will be explained in this section.

The constraints used by Co-op are based on the ones presented by Nagy [69]. Co-op provides two types of constraints: ordering and control constraints. Ordering constraints influence the execution order only, they do not influence what will be executed. Control constraints do the complement: they express what will be executed, without influencing the execution order. Nagy does not make this distinction, his control constraints can influence the execution order and vice versa. We explicitly decoupled these two concerns, because we think it is better to address them separately.

Binding presence and success are closely related, as shown in section 2.3. Therefore, we have chosen the constraints to have the same outcome on binding absence and failure. To achieve this, we use—from the constraints presented by Nagy—as ordering constraint the soft pre and as control constraints the hard $cond$ and hard $skip$. This results in the following constraints:

- $p_pre(A, B)$ only allows the execution of B when A has been executed already or will not be executed at all.

- $cond(A, B)$ only allows the execution of B when A is applicable.
- $p_skip(A, B)$ only allows the execution of B when A is not applicable.

TRANSITIVITY Some constraints are not transitive. For example, p_skip is not transitive: $p_skip(A, B) \wedge p_skip(B, C)$ does allow the execution of A and C or the execution of B (or any subset thereof). In general, $skip$ is used to specify overriding relations, which means that only a single method should be executed (see Te Brinke [P.1] for more details). A transitive $skip$ constraint specifies such an overriding relation. Therefore, we consider transitivity a desired property and call the non transitive constraints *primitive*. This is denoted by using the prefix $p_$ before the names of non transitive constraints.

COND The transitivity of $cond$ follows from its definition. Assume that we have defined $cond(A, B)$ and $cond(B, C)$, then transitivity requires that $cond(A, C)$ holds. Only when A is not applicable and C is applicable, $cond(A, C)$ would not be satisfied. Therefore, we consider only situations where A is not applicable. When A is not applicable, B cannot be applicable, because of $cond(A, B)$. Similar, C cannot be applicable, due to $cond(B, C)$. Thus, when A is not applicable, C is not applicable either. This means that $cond(A, C)$ holds, so we can conclude:

- $cond$ is transitive: $cond(A, B) \wedge cond(B, C) \Rightarrow cond(A, C)$

PRE For pre , this is different. By itself, p_pre is not transitive. For example, when we define $p_pre(A, B)$ and $p_pre(B, C)$ and B is absent, $p_pre(A, C)$ might not hold. A possible execution order would be: $\langle C, A \rangle$. This execution order satisfies both defined constraints, but not the constraint implied by transitivity.

However, we have defined pre to be transitive. This transitivity can be ensured by calculating the transitive closure over all pre constraints. When this is done, there is no need to add any additional kind of constraint; every pre constraint can be handled as a primitive one. The result is:

- pre is transitive: $pre(A, B) \wedge pre(B, C) \Rightarrow pre(A, C)$

SKIP As we saw earlier, p_skip is not transitive, which might result in undesired behavior. Therefore, we also provide a transitive version of $skip$.

- $skip$ is transitive: $skip(A, B) \wedge skip(B, C) \Rightarrow skip(A, C)$

In total, we have defined five constraints, all listed in table 6. However, we have only three basic kinds of constraints, because the transitive pre and $skip$ can be defined in terms of the primitive version of these constraints.

When there are conflicts between constraints, it can be impossible to identify which bindings should be executed in what order. An example is when $pre(A, B)$ and $pre(B, A)$ are both present. In such a case, an exception will be thrown.

Constraint	Meaning
$p_pre(A,B)$	Prioritizes execution of binding A over execution of binding B.
$pre(A,B)$	Idem, but transitive.
$cond(A,B)$	Allows execution of binding B only if binding A is applicable.
$p_skip(A,B)$	Skips execution of binding B if binding A is applicable.
$skip(A,B)$	Idem, but transitive.

Table 6: Constraints

Figure 5: Example of the $skip(\text{defaultBinding}, \text{virtualBinding})$ constraint

Constraints Handle Binding Instances

We have now seen the intention of the different constraints. However, because constraints are expressed between binding classes, they have to handle a set of binding instances at runtime. Thus, in order to define the semantics of constraints more precisely, we have to take into consideration that they work over sets.

For example, in figure 5 are two instances of the virtual binding and a single default binding. Between the two binding classes, the following constraint is defined:

```
skip(defaultBinding, virtualBinding);
```

Using this notion of a set of binding instances, we define the precise semantics of constraints as follows:

- $pre(A, B)$ only allows the execution of $\beta \in B$ when $\forall_{\alpha \in A} \alpha$ is applicable $\Rightarrow \alpha$ has been executed already.
- $cond(A, B)$ only allows the execution of $\beta \in B$ when $\exists_{\alpha \in A} \alpha$ is applicable.
- $skip(A, B)$ does not allow the execution of $\beta \in B$ when $\exists_{\alpha \in A} \alpha$ is applicable.

The semantics of $pre(A, B)$ are also shown in figure 6. It shows that all elements from A, the left set, should have been executed before any element of B, the right set, can be executed. For $cond(A, B)$ and $skip(A, B)$ we see in figures 7 and 8 that the applicability of any element from A is enough to respectively allow or cancel the execution of all elements of B.

Constraint Activation

Only activated constraints are used by the runtime. Constraint activation is the same as binding activation. An example is the following the constraint:

Figure 6: Working of pre constraint

Figure 7: Working of cond constraint

Figure 8: Working of skip constraint

```
constraint inheritanceConstraint = skip(defaultBinding, virtualBinding);
```

This constraint can be activate by calling:

```
inheritanceConstraint().activate();
```

Deactivation works similar:

```
inheritanceConstraint().deactivate();
```

In general, it is desired to activate the constraints before activating the bindings of a composition operator. Constraints have no influence on the runtime when the bindings they are specified on are not activated. However, bindings might cause unintended behavior when activated without activation of the desired constraints.

3

COMPOSING BEHAVIOR

This chapter is based on our publication “Free Composition Instead of Language Dictatorship” [P.6], which was presented at the 7th International Conference on Software Paradigm Trends. It explains the idea behind Co-op—why is a language with first-class composition operators needed?—and introduces the simplest kind of composition: composing behavior.

To simplify the development and maintenance of energy-aware software, the designer should be able to separate the energy and functional concerns in particular. For a programming language that facilitates both separating concerns into modules and composing such modules in general, we define the following two requirements:

1. Expressivity
2. Compositionality

By expressivity we mean that the implementation language must directly express the functional part. Otherwise, workarounds are needed for implementing the basic functionality, which increases complexity and reduces reuse.

By compositionality we mean that the language provides means to semantically and syntactically integrate energy aware parts and functional parts, such that these concerns can be developed separately and utilized together. Otherwise, the resource concern becomes tangled with the functionality, which results in increased complexity.

3.1 COMPOSITION

Decomposing software into modules is not enough to achieve modular software. For a system to work well, the relations among modules require as much design and development attention as the modules [90]. If modules are not composable, the relations among modules become complex. Thus, programming languages must support *composable* modules. Composition exists in several variants:

- Behavior, such as method calls and behavior composition through traits.
- Data, such as variable access and object composition through inheritance.
- Control flow, such as loops and exception handling.

3.1.1 Examples of Explicit Composition Mechanisms

Programming languages¹ offer explicit and implicit means to express composition of abstractions, which means that the characteristics of a composed abstraction are expressed in terms of the characteristics of one or more existing abstractions (and possibly additional specifications). For example, *function composition* in functional programming, such as $f(g())$, expresses that—the behavior of— f and g are composed by using the result of g as an argument of f . Similarly, *function invocation*, such as the call to q in $p()\{p_1; q(); p_2\}$ is used to define part of the behavior of a procedure p in terms of the behavior of q . In this sense, the function invocation expresses that the behavior of p is a composition of the behavior of p_1 , $q()$, and p_2 . Another example is *object aggregation* (also referred to as *object composition*): this expresses how a new object structure is defined as, e.g., the union of other object structures.

In typical object-oriented languages, the following composition mechanisms are available: behavior composition through message passing (cf. function invocation), object aggregation, and inheritance. The simplest compositions are binary, such as function invocation and single inheritance, which compose two functions or two classes, respectively. However, n-ary compositions that involve multiple components are also important [90]. Examples are multiple inheritance and pointcut-advice composition in AOP [58], which compose multiple classes or multiple advices, respectively.

3.1.2 Implicit Composition Mechanisms

Composition is not necessarily implemented *directly* by language constructs; for example, many design patterns [43] describe how a set of objects interact to create a coherent new behavior: a composition of the participating objects. This composition is typically realized by a set of code snippets distributed over the participating objects, which must be re-implemented for every design pattern instantiation [73, 99]; we refer to these code snippets as glue code. Patterns can be seen as application templates or mini architectures. Their advantage is that patterns can be expressed—through glue code—in existing languages, whereas other composition mechanism require language constructs. However, glue code has several disadvantages:

- It obfuscates the design; instead of specifying the relation of two or more modules explicitly, glue code defines the composition imperatively. Because glue code is scattered over multiple participating modules, the design intention becomes even more implicit.
- It is difficult to develop and maintain. Although patterns are in general not complicated, their correctness may not be easily enforced, owing to the distribution of code duplicates over several locations.

¹ Whenever we talk about languages, this should be interpreted broadly to any means of specifying machine-executable behavior.

3.2 PROBLEM ANALYSIS

Historically, programming languages have been—benevolent—dictators: reducing all possible semantics to specific ones offered by a few built-in language constructs. Over the years, some programming languages have freed the programmers from the restrictions to use only libraries, data types, and type-checking rules that are built into the language. Even though—arguably—such freedom could lead to anarchy, or people shooting themselves in the foot, the contrary tends to be the case: a language that does not allow for extensibility is depriving software engineers of the ability to construct proper abstractions and to structure software in the most optimal way. Software would become more structured and maintainable if the software engineer could express the behavior of the program with the most appropriate abstractions.

The history of programming shows a steady movement towards supporting higher-level abstractions of building blocks and more advanced ways of expressing compositions. For example, object-oriented and aspect-oriented programming are motivated by the need for improved modularity and separation of concerns; trends in software engineering, such as Model-Driven Engineering (MDE) and Domain-Specific Languages (DSLs), all aim at offering an appropriate abstraction level for expressing particular types of problems. To this extent, they offer *(a)* dedicated (possibly graphical) syntax, *(b)* dedicated data types and operators, or *(c)* dedicated abstractions and corresponding composition techniques, to achieve better modularity and separation of concerns for specific domains.

New composition mechanisms are introduced all the time. For example, Taivalsaari [95] describes a taxonomy for inheritance mechanisms, from which—in theory—hundreds of variants for inheritance can be derived. More recently, many proposals have been made for aspect-oriented languages and models: a survey report [15] contains 45 different proposals, where in most cases the composition techniques are unique. Similarly, there is a large number of design patterns that all express a composition of objects that cannot be expressed directly in mainstream programming languages.

In general, it can be safely assumed that for each of these proposed composition mechanisms, there is a sound argumentation why that mechanism works better—at least for a certain class of applications, or within a certain context. The fundamental reason is that the application of each composition mechanism involves a certain trade-off, which makes it particularly suitable in certain contexts, but less so in others. Hence, a language that dictates a fixed set of composition techniques, with no opportunity to extend that set, will inherently restrain the software engineer: he or she is not able to choose the most appropriate composition mechanism, with the best possible trade-offs, and the most natural mapping from a conceptual solution to the implementation. Among the negative results caused by such dictatorship² are

LACK OF DECLARATIVENESS because it does prevent the developer from directly specifying the composition without using workarounds.

² There are in fact also positive sides; e.g.: less choice makes both decisions and comprehension easier—albeit at substantial costs.

LACK OF TRACEABILITY because the intended composition has to be replaced with a work-around, typically involving additional ‘glue code’.

LACK OF MAINTAINABILITY because the glue code is usually specific to the context, and has to be added in multiple locations, where it is also tangled with the functionality.

INCREASED ACCIDENTAL COMPLEXITY because straightforward compositions at the conceptual level have to be realized by more complicated code that introduces additional dependencies.

A language that offers mechanisms to implement libraries of composition operators facilitates language experts to define new compositions and hide the complexity of their implementation behind an API or library. Regular programmers can simply use the composition operators without being bothered with their implementation or the facilities offered by the language specific for implementing composition operators; programmers can simply instantiate composition operators as needed.

3.3 SCOPE AND ASSUMPTIONS

Our approach allows free composition in contrast to languages that limit composition to a few mechanisms. We propose language facilities where the appropriate composition mechanisms can be defined, applied, and reused. Before explaining this approach in more detail, we first discuss the scope of our solution approach and our assumptions. This discussion should also help to distinguish our work from various areas of related work.

Although not essential to the general idea, we focus on object-based languages, which support encapsulation and message passing. In this context, the first-class entities are *objects*, so composition refers to *composition of objects*.

Our approach is *not* based on full reflection: although that is a very powerful technique, reflection-based solutions tend to be very hard to organize and manage and, hence, are not suitable from a scalability point-of-view. In particular, building fully reflective solutions that are also composable and extensible, requires a substantial amount of effort and discipline for the involved software engineers [57]. However, we do propose to adopt limited reflection (where only specific elements of a language are exposed for reflection). We would like to point out that our approach can well be implemented, *using* reflection, as a specific Meta-Object protocol (MOP) [59].

We also aim for solutions that are not transformation based, such as most Model-Driven Engineering (MDE) tools and external Domain-Specific Languages (DSLs). Although implementing composition operators as transformations is in principle a possibility, this suffers from several issues. In particular, if multiple composition operators are to be integrated within a larger solution: (a) it is hard to exchange data between the model (or DSL) world and the rest of the system, (b) it is hard to add additional DSLs without running into all kinds of integration issues, and (c) it requires additional tool support to develop at the model level. For example, after transforming (or compiling) a specific DSL to the common core, its original structure is lost, which increases the complexity of debugging [107]. Bidirectional trans-

Figure 9: Example of the composition operator for inheritance

formations can avoid such loss of information, but specifying these still has considerable complexity [80, 92].

Libraries are formidable competitors to DSLs [67], and can be referred to as domain-specific embedded languages (DSELs) [54]. Such embedding is a simple way to construct well-designed languages for specific application areas, according to Kamin [56]. He also points out that general programming features *are* needed, but often omitted from external DSLs: “The beauty of language design by embedding is that the programming features come automatically and for free.” Mernik [67] concludes that GPLs should provide more powerful support for embedding DSLs. This is exactly what we aim for: a GPL that allows composition operators to be embedded through libraries.

Windhouwer [T.7] showed that it is possible to implement RUMs in Python [85], a language that provides both meta classes (MOP) and aspect-oriented (AOP) features [66]. However, he also shows that fully supporting RUMs reduces the readability of the source code.

Bouius [13] tried to introduce energy-awareness in existing cyber-physical system without crosscutting existing concerns. He concludes that energy-awareness functionality can be decoupled from most components, but it *cannot* be modularized into a single component, when using object-oriented or aspect-oriented programming languages.

Thus, we can conclude that better composition mechanisms are needed for implementing RUMs in a modular way. To improve the programming language landscape in a way that clean implementations of RUMs are facilitated, our idea is to move composition from built-in language constructs to programmable, first-class abstractions in a language. We designed the Co-op language, which shows that it is possible with a relatively simple model, to express a wide range of compositions as first-class concepts. Co-op was largely designed before we designed the RUM and provides a strong composition mechanism needed for implementing RUMs, but studying its suitability to support RUMs in more detail is still future work.

3.4 GENERAL APPROACH

The general approach of *first-class composition operators* (Co-op) was first introduced by Havinga [50, 51], and will be explained in this section. We have continued the work of Havinga and present our work in the following sections.

In Co-op, composition operators are first-class entities (i.e.: objects) that *operate* on compositions. An example is the composition operator *inheritance* (shown in figure 9a), which

Figure 10: Composition operators reroute messages

composes the behavior of two classes (as in figure 9b) such that their behavior is combined. That is, the subclass provides the composed behavior of both classes (see figure 9c).

Composition operators can (cooperatively) influence the behavior of a composition. As we will discuss, this approach is sufficiently powerful to express common composition mechanisms such as inheritance, delegation, and design patterns. While the Co-op approach is not specific to a single language, it assumes that the underlying language has a concept of *objects* (i.e.: first-class entities, commonly without built-in support for inheritance) and *message passing* between objects. These concepts can express the composition operator *inheritance* as shown in figure 10: whenever a message is sent to an object that inherits the implementation from its super class (e.g.: the method `talk` is called on a `Student`), this message is rerouted to the implementation provided by the super class.

Co-op stipulates the following language constructs for defining and controlling composition operators:

- Only one primitive composition operator is provided by the language, which is *sending a message*. Other composition operators, such as inheritance, are not manifested in the language's syntax.
- Composition operators are first-class entities in the language: we believe this is an essential feature for a scalable approach (see also composability (3) below). This also means that the application of composition operators can be as simple as object instantiation, and composition operators can be managed as regular libraries.
- Composability (1): it is possible to freely apply multiple composition operators within the same application. For example, Smalltalk-style [45] and Beta-style [62] inheritance can be used for different classes in the same program. (The difference between these styles is that in Smalltalk the lookup starts in the subtype and is bottom up, whereas in Beta it starts in the super type and is top down.)
- Composability (2): it is possible to combine multiple composition operators in the same context, i.e.: apply them to the same objects. For example, inheritance and delegation can be combined [10].
- Composability (3): composition operators can be used in the definition of other composition operators (as long as this does not cause infinite recursion). Bergmans, Havinga, and Akşit [10] show how the two composition operators *PointcutAdvice* and *AspectJ-PointcutAdvice* are composed through the use of another composition operator: *SingleInheritance*.

Figure 11: Overview: Composition in *Co-op*

- Inherent dependencies or constraints among composition operators are expressible, such as exclusion, relative ordering, and overriding among composition operators. For example, Smalltalk and Beta-style inheritance are mutually exclusive. When combining inheritance with delegation, ordering matters: inheritance could have precedence over delegation, or the other way around.

Chapters 4 and 5 illustrate how the Co-op approach can be used to implement composition operators such as composition of data access and control flow, which are needed to compose a RUM with a functional component.

3.5 CO-OP

While the syntax for a *message send* is embodied in a language that follows our approach, the semantics of where a message is delivered is not fixed, but determined during dispatch by the composition operators in Co-op (see figure 11). A message send is, by itself, undirected; i.e., when the program sends a message, it specifies property values, such as the message name or initial argument values. Composition is performed at message sends, by rewriting the message's properties; since all message properties may be rewritten by composition operators, e.g., the initial target is not necessarily the receiver of the message.

Just like in other object-based languages, *Co-op classes* are templates for *objects*. Classes specify the structure (available fields) and code (method bodies) of objects, whereas objects encapsulate state (values of fields) and behavior. In addition, Co-op classes may also specify composition operators, which can extend or change the behavior of objects. Figure 12 shows the composition infrastructure schematically. In this example, the object *bmy* delegates all its behavior to the object *stb*. In the lower left, the object *lw* performs a *message send*. The reified message is evaluated by the active set of so-called *bindings* (which will be detailed below), defined by *Co-op classes*. Finally, the evaluation of a message send typically leads to the invocation of one (or more) method(s) on an object. Here, the method *study* of the object *stb* is invoked.

The key characteristic is that all possible behavior involves message dispatch; hence manipulating message dispatch can be used to express a truly wide range of behavioral compositions. The semantics of a particular composition technique are specified by one or more of the following three additional class members:

Figure 12: Composition in *Co-op*: Delegation

1. *Conditions* define Boolean predicates that refer to message properties and to fields or methods defined in the class. For example, the *delegationBinding* selects all messages that are targeted to the delegator *bmy*.
2. *Bindings* define the conditions to specify which messages are selected to participate in the rewriting of a message, and assign new values to message properties when they are applicable; hence they bind source messages to new target messages. The target messages may be directly executable, or may be manipulated once more by other bindings. For example, the *delegationBinding* selects all messages based on the condition shown at 1 and reroutes these messages to the delegatee by rewriting the target to *stb*.
3. *Constraints* define relations between composition operators with bindings that are applicable at the same message send. Relations are, for example, the order of evaluation and the prevention of evaluating one operator. This can express exclusion, ordering, and overriding of compositions. Here, there is a *skip* constraint between the *delegationBinding* and *defaultBinding* to ensure that whenever the behavior is delegated, it is only performed by the delegatee; the delegator will skip this behavior.

Havinga et al. [10, 51] defined an operational semantics and implemented an interpreter for a first prototype of Co-op, called Co-op/I, in Java. Co-op/I faithfully realizes the execution approach described in this chapter and exposes all method invocations as *message sends*. In this prototype language, many different composition operators such as aspects, delegation, traits, various forms of behavioral inheritance [51], and several design patterns [52] have been realized.

4

COMPOSING DATA ACCESS

This chapter presents how behavior can be composed inside Co-op, based on our publication “The Keyword Revolution: Promoting language constructs for data access to first class citizens” [P.2], presented during the 1st International Workshop on Free Composition.

The previous chapter showed how behavior can be composed inside Co-op. However, composition techniques such as inheritance do not only control the composition of behavior, but also the composition of data. For instance, access modifiers control from where data fields can be accessed. For example, only by method definitions contained in the same class as the field declaration; or also by methods defined in classes inheriting from the one declaring the field. RUMs collect data from various locations and combine this data for optimization, so data composition is also relevant for RUMs.

We implemented a second prototype of a Co-op language and execution environment, Co-op/II [P.1]. In this prototype, additional to function calls, also data accesses are reified as messages being sent and composition operators can reason about and influence such messages. This chapter explains the concepts of Co-op in terms of the Co-op/II prototype. All code examples can be executed by the Co-op/II interpreter¹.

4.1 KEYWORD FREE COMPOSITION OPERATORS

In most (OO) programming languages, there are many keywords and fixed language constructs to manipulate the way that data is, or can be, accessed. We mention a few examples:

- Access modifiers in Java, C++ and C# are *public*, *protected*, and *private*. The language C++ adds a *friend* keyword to express yet another form of access rights on data (as well as behavior). Note that there is a wide range of possible access modifiers, when including the notion of package-level protection, or the distinction between class-level and instance-level protection.
- The Java, C++ or C# keyword *static* controls whether all instances of a class share a field, or each has its own copy.
- The keywords *final* in Java, *const* in C++, and *readonly* in C# declare special semantics to the usage of the variable (i.e., the variable may be assigned only once).

For every new keyword or feature that is introduced in a language, it must be considered carefully how this interacts with all possible combinations of other language constructs. This can be very challenging, and also tends to make the evolution of the language over

¹ See appendix A and the SourceForge project *Co-op* at co-op.sf.net.

time very difficult. As a result, creating new language constructs to address all the desired features in one language is not feasible.

Our proposal, as illustrated in chapter 3, is to aim for a simple object model, and a single mechanism for expressing a wide range of behavioral modifiers for fields. Examples of modified composition semantics, which can be expressed with the proper composition operators in Co-op/II, are access modifiers, static, synchronized, final, and so forth, but also more conceptual constructs such as automatic conversions, checking of validity constraints, persistence, transactions, or expressing roles. Composition operators supporting these semantics can be provided by reusable libraries.

4.2 USING A COMPOSITION OPERATOR: INHERITANCE

Inheritance is a common technique for composing classes. In practice, composing classes means composing all objects that are instances of these classes. Here, we use the Smalltalk inheritance style to illustrate the implementation of a composition operator. Smalltalk inheritance allows for overriding methods along a single inheritance hierarchy. As an example, we introduce the following class as a subclass of class Person:

```

1 class Student {
2   method study() {
3     return "Is that necessary?";
4   }
5 }
```

The inheritance relation is not specified through a built-in language construct, but by composition operators, as shown in the following code, which could be part of the main() method of the application:

```

1 SmalltalkStyleInheritance.subclassOf(Student, Person);
2 var alice = Student.new("Alice", "Bachelor");
3 System.println( alice.study() );
4 System.println( alice.talk() );
```

In the case of inheritance, it may be more intuitive (and in fact, appropriate), to specify inheritance within the subclass itself; this can be achieved in Co-op either by writing it in the class initializer of Student, or to express it through an annotation on the class declaration, such as `class @Inherits(Person)Student`, but the latter is not supported by our prototype. The key point of our contribution is that the inheritance specification is no longer part of the language syntax, and the location where to specify it can now be determined to provide the best design trade-off.

4.3 DEFINING A COMPOSITION OPERATOR: INHERITANCE

The inheritance behavior in section 4.2 is in fact composed of two composition operators: one for *method inheritance* (behavior), and one for *field inheritance* (data). These are created

in respectively lines 3 and 4 of the following listing. The definition of `FieldInheritance` takes a third parameter, here class `LocalFieldAccess`, which defines the policy for field access. To express Smalltalk-like inheritance, we want fields to be accessible only locally, and not from subclasses:

```

1 class SmalltalkStyleInheritance {
2   method @ImplicitParameters([]) subclassOf(childType, parentType) {
3     MethodInheritance.subclassOf(childType, parentType);
4     FieldInheritance.subclassOf(childType, parentType, LocalFieldAccess);
5   }
6 }

```

4.3.1 Defining Inheritance of Methods

This subsection illustrates how method inheritance can be expressed using Co-op/II (which is similar to that in Co-op/I). The execution of `alice.talk()` (i.e., invoking an inherited method) involves two dispatch steps:

Alice is an instance of `Student`, and the `talk()` method is implemented in `Person`, so Alice cannot execute this behavior directly. To express inheritance, the `virtualBinding` rewrites messages sent to a `Student` to address a `Person`. That is, it rewrites the `targetType` property of the message from `Student` to `Person`, after which the `defaultBinding` is applicable and invokes the method `talk` implemented by the class `Person`. The composition operator that specifies this binding is defined as follows:

```

1 class MethodInheritance {
2   var childType;
3   var parentType;
4
5   // Binding for virtual method lookup
6   binding virtualBinding = (messageKind == "Call" & targetType == this.childType) {

```

```

7   targetType = this.parentType;
8   }
9
10  // virtualBinding is applicable only if the default binding fails
11  constraint bottomUpResolution = skip(defaultBinding, virtualBinding);
12
13  // initialize the instance variables and activate binding.
14  method subclassOf(childType, parentType) {
15    this.childType = childType;
16    this.parentType = parentType;
17    this.bottomUpResolution().activate();
18    this.virtualBinding().activate();
19  }
20 }

```

Each instance of `MethodInheritance` defines an inheritance relation between the `childType` (line 2) and `parentType` (line 3). The `virtualBinding` in this example selects all calls to the `childType` (line 6) and reroutes these to the `parentType` (line 7), effectively specifying virtual method lookup. This virtual lookup only takes place if regular lookup (i.e.: the `defaultBinding`) fails, specified by the constraint `bottomUpResolution` (line 11): if the default binding succeeds, the `virtualBinding` is skipped.

The presented `MethodInheritance` achieves Smalltalk-style inheritance, where the inheritance works bottom up: the method lowest in the inheritance hierarchy is executed. For Beta-style inheritance, the lookup works top down. This can be achieved in Co-op by replacing the given constraint by the following one:

```

9   constraint topDownResolution = skip(virtualBinding, defaultBinding);

```

4.3.2 Defining Inheritance of Fields

Defining the availability and accessibility of fields from superclasses is needed for expressing various inheritance (and other composition) semantics. The first version of the Co-op language, Co-op/I, did not support this.

For example, the Smalltalk-style inheritance we have defined earlier, also instantiates the `FieldInheritance` composition operator. Even though the fields of a `Person` are not directly accessible from its subclass `Student`, they are still addressable through methods defined in the class `Person`. Thus, each instance of `Student` must contain all state of a `Person` in addition to its own state. Therefore, upon creation of a child class, also all fields in its superclasses must be created. That is what the composition operator `FieldInheritance` does.

`FieldInheritance` also creates a relation between an instance of the child class, and an instance of its superclass². For Smalltalk-style inheritance, this relation is created using the `LocalFieldAccess` operator, which allows private field access only.

Now, consider you do not want field access to be limited to the declaring class only, but you want them to be also addressable through the child classes. For example, allowing us to write the following implementation of `Student`:

² This does not involve specific and optimized memory layouts for objects.

```

1 class Student {
2   method study() {
3     return "Is that necessary for " + this.title + " " + this.name + "?";
4   }
5 }

```

The field accesses in the method `study()` address fields defined in the super class, just like the way we can address methods defined in the super class. Therefore, a binding that realizes accessing fields through child classes can be defined in a similar way to defining it for methods:

```

1 class InheritedFieldAccess {
2   var child;
3   var parent;
4   // Binding for field lookup in parent type
5   binding inheritFields = (messageKind == "Lookup" & target == this.child) {
6     target = this.parent;
7     targetType = System.classOf(this.parent);
8     this = this.parent;
9   }
10  // inheritFields is applicable only if the default binding fails
11  constraint bottomUpResolution = skip(defaultBinding, inheritFields);
12  // initialize instance and activate binding
13  method initDispatch(child, parent) { ... }
14 }

```

The most notable difference with `MethodInheritance` is that in this case we have a parent and child *instance* (lines 2 and 3) instead of *type*, because field values are instance properties, whereas method behavior is a property of the type. This also means that the binding selects calls to the child *instance* (line 5) and reroutes these to the parent *instance* (lines 6 to 8).

Now, we can use field inheritance as shown in the example below. First, we create a relation between the parent and child type and then we can access all fields of the parent type also through any instance of a child type, which the method `study()` does.

```

1 MethodInheritance.subclassOf(Student, Person);
2 FieldInheritance.subclassOf(Student, Person, InheritedFieldAccess);
3 var alice = Student.new("Alice", "Bachelor");
4 System.println( alice.study() );

```

Languages such as Java and C++, which disable inheritance of fields by default, also provide mechanisms to let the developer select fields that are accessible through subtypes. We have modeled this using the `SelectiveFieldInheritance` composition operator, which can replace the `InheritedFieldInheritance`. This operator only enables inheritance for fields that are selected explicitly. Its use is as follows:

```

1 MethodInheritance.subclassOf(Student, Person);
2 var selectiveInh = SelectiveFieldInheritance.new();
3 selectiveInh.initDispatch(Student, Person);
4 selectiveInh.addField("name");
5 var alice = Student.new("Alice", "Bachelor");
6 System.println( alice.study() );

```

Since this example allows subclasses of `Person` only access to the field `name`, the implementation of `Student` shown in the beginning of this section will yield a runtime error. However, the following implementation of `Student` is correct when this selective inheritance is applied:

```

1 class Student {
2   method study() {
3     return "Is that necessary for " + this.name + "?";
4   }
5 }

```

Traditional programming languages allow programmers to add access rules to fields or methods by adding a modifier to their declaration. To enable a similar programming style, Co-op allows the addition of annotations to field and method declarations. Using reflective capabilities to access these annotations—which are not yet possible in the Co-op/II prototype—allows implementing the `FieldInheritance` composition operator in such a way that it, for example, only applies to fields with the `@Inherited` annotation:

```

1 class Person {
2   var @Inherited name;
3   var title;
4 }

```

4.4 PROVIDING A COMPOSITION LIBRARY: INHERITANCE

Composition operators can be provided by reusable libraries. We provide such a library containing 13 of the 16 alternative semantics for inheritance presented by Taivalasaari [95], which we will show after introducing the inheritance classification of Taivalasaari.

4.4.1 Inheritance Classification of Taivalasaari

A detailed description of inheritance is presented by Taivalasaari [95]. In general, an inheritance hierarchy is a directed acyclic graph (DAG). A simple example of an inheritance DAG is shown in figure 13. Which DAGs represent valid inheritance hierarchies depends on the kind of inheritance. According to Taivalasaari, inheritance has eight main characteristics. This section will give a short description of these different characteristics of inheritance.

INHERITANCE TYPE Most object oriented systems are built around *classes*, which are blueprints from which instances can be created. Another possibility are prototype based systems. These systems have no classes, but are built around *objects*. These objects are prototypes providing default behavior. Instances are created by cloning already existing prototypes. Inheritance can be achieved by modifying an individual instance, which then becomes a new prototype that can be cloned.

SHARING TYPE To share common behavior between objects, the relation between these objects must be expressed. Within computer memory, there are only two ways to express

Figure 13: Basic inheritance DAG

Sharing type	Delegation	Concatenation
Record combination strategy	sharing/references	copying/contiguity
Interface dependence	dependent (life-time sharing)	independent (creation-time sharing)
Inheritance DAG	preserved	flattened

Table 7: Delegation versus concatenation

direct relationships: references and contiguity. This results in two elementary strategies for inheritance: *delegation*, which uses references, and *concatenation*, which uses contiguity. Often, common behavior is shared by using delegation: when an object receives a message it does not understand, it will delegate this message to those objects that have been designated as its “parents”. It is also possible to capture the essence of inheritance by using concatenation instead of delegation. Concatenation can be done by utilizing cloning together with the ability to add new properties to objects dynamically. An overview of the differences between the two sharing types is given in table 7.

VERTICAL COMBINATION In most main-stream object oriented languages, lookup proceeds from the most recently defined (descendant) parts of the object to the least recently defined parts (parents). This lookup order is called *descendant-driven* inheritance. Another possibility, adopted by Beta, is *parent-driven* inheritance, which traverses the inheritance DAG in the opposite direction. Figure 14 illustrates these lookup directions. The inheritance direction is the same in both situations—P extends C. However, the start and direction of lookup and also the direction of the self reference are opposite.

Besides the difference in direction, vertical combination can also differ in the completion of method lookup. When the same method is defined as part of multiple classes in the inheritance DAG, calling the method can be handled in multiple ways. Simplest is disallowing this situation, which disallows any overriding, called *strict*. The most frequently adopted approach is *asymmetric*: terminating the lookup immediately after encountering the first matching method. It is also possible to execute every matching method. This *composing* completion is, for example, used by Beta.

When combining the different types of direction and completion, we have five variations of vertical combination, which are shown in table 8. The inheritance hierarchy for the column $x \in C \wedge x \in P$ is shown in figure 15. In this case, x is defined as a member of C and

Figure 14: Both directions of lookup

Figure 15: Vertical overlapping

Figure 16: Horizontal overlapping

P , causing vertical overlap because P is an ancestor of C . The other two columns represent the situations without vertical overlap; here x only is a member of C or P respectively.

HORIZONTAL COMBINATION When the inheritance DAG contains overlapping properties along different paths, these properties are horizontal overlapping. An example of such a hierarchy is shown in figure 16. By using *ordered* horizontal combination, an order is defined among the different paths. This allows the execution of the first matching property. When no such order is defined, i.e.: the horizontal combination is *unordered*, the presence of horizontal name collision is an error.

C extends P	$x \in C \wedge x \notin P$	$x \in C \wedge x \in P$	$x \notin C \wedge x \in P$
Strict	$C.x$	error	$P.x$
Asymmetric descendant-driven	$C.x$	$C.x$	$P.x$
Asymmetric parent-driven	$C.x$	$P.x$	$P.x$
Composing descendant-driven	$C.x$	$C.x \circ P.x$	$P.x$
Composing parent-driven	$C.x$	$P.x \circ C.x$	$P.x$

Table 8: Variations of vertical lookup combination

MULTIPLE INHERITANCE Single inheritance, for example provided by Java, only allows extending a single class. Multiple inheritance, for example provided by C++, also allows extending multiple classes. Thus, single inheritance limits the inheritance hierarchy to trees, whereas multiple inheritance allows the hierarchy to be any DAG.

DYNAMIC INHERITANCE In general, the inheritance structure is defined statically. However, dynamic inheritance provides the ability to change parents dynamically at runtime.

SELECTIVE INHERITANCE Not passing all information from the parent to the child is called selective inheritance. Two cases of selective inheritance can be distinguished: *selective attribute inheritance* and *selective value inheritance* [105]. Selective attribute inheritance allows inheriting only a subset of the members of the parent object. Selective value inheritance provides the ability to inherit a subset of the values of the parent object. Thus, selective value inheritance is a kind of attribute inheritance where some values are the same for the parent and child object, whereas others—the ones not using value inheritance—might differ. Value inheritance is mainly useful when combined with object inheritance, since it allows choosing whether the values of the child and parent object can be updated independently or will remain equal.

MIXIN INHERITANCE Mixin inheritance provides dedicated mixin classes. Such mixin classes are syntactically equivalent to normal classes, but have a different intention. A mixin class is defined solely for adding properties to other classes. Mixins are never instantiated and have no superclasses. By combining a mixin with a base class using multiple inheritance, the functionality of the mixin is added to the base class.

It is possible to combine each value for one characteristic with all other characteristics. However, some combinations might not be useful. For example, when using single inheritance, horizontal combination is not important, because only one parent is available. Which combination of characteristics is provided depends on the programming language. A few examples are: Smalltalk provides class-based, delegation based, asymmetric descendant-driven inheritance. Beta provides class-based, concatenation based, composing parent-driven single inheritance. Java provides class-based, delegation based, asymmetric descendant-driven single inheritance.

4.4.2 Composition Library

Composition operators can be provided by reusable libraries. We provide such a library for Co-op/II, containing 13 of the 16 alternative semantics for inheritance presented by Taivalsaari [95]. Table 9 shows which semantics are part of the Co-op library and compares it with the programming languages Java, Beta, C++, and Smalltalk. The semantics that are not supported by our prototype are shown as checkable squares, because anyone could provide an implementation of such a composition operator. This is a major difference with the other languages listed, because these do not offer facilities for adding the missing semantics. In our examples, we reuse the implementation of lower-level composition operators by combining them to more complex ones, again by means of Co-op/II composition operators.

Property	Co-op	Java	Beta	C++	Smalltalk
Inheritance type					
Class based	✓	✓	✓	✓	✓
Object based (prototype based)	✓	—	—	—	—
Sharing type					
Delegation	✓	✓	—	✓	✓
Concatenation	□	—	✓	—	—
Vertical combination					
Strict	✓	—	—	—	—
Asymmetric descendant-driven	✓	✓	—	✓	✓
Asymmetric parent-driven	✓	—	—	—	—
Composing descendant-driven	✓	—	—	—	—
Composing parent-driven	✓	—	✓	—	—
Horizontal combination					
Ordered	□	—	—	✓	—
Unordered	✓	—	—	—	—
Multiple inheritance	✓	—	—	✓	—
Dynamic inheritance	✓	—	—	—	—
Selective inheritance					
Selective value inheritance	□	—	—	—	—
Selective attribute inheritance	✓	—	—	—	—
Mixin inheritance	✓	—	—	—	—

- ✓ Supported by the programming language
- Could be implemented in the programming language
- Not supported by the programming language

Table 9: Supported inheritance properties by various languages

Compared to Co-op/I, we mainly add the ability to compose behavior. In Co-op/I, fields can only be referred through their declaring class, so fields are effectively private. Therefore, composition operators in Co-op/I only influence behavior. The composition library of Co-op/I provides asymmetric descendant-driven, composing parent-driven, and ordered multiple inheritance [50, 52]. However, these implementation are limited to behavior, whereas our implementations in Co-op/II influence as well data as behavior.

Table 9 shows that Co-op already supports much more than mainstream languages. However, our Co-op library does not provide the following three semantics of inheritance: sharing by concatenation, ordered horizontal combination, and selective value inheritance. The next three paragraphs provide a short explanation why we did not implement these semantics in Co-op. More details on the implementation of the library can be found in appendix B.

CONCATENATION The sharing type identifies how relations are expressed in computer memory. However, Co-op does not provide access to the computer at a low level. Thus, it is not possible to influence the memory directly. Therefore, it is not possible to use concatenation as sharing type.

According to Taivalsaari, delegation and concatenation achieve the same result, but the way in which message sending is handled differs. Concatenation supports independent modification of object interfaces (see table 7), whereas delegation provides dependent interfaces. Therefore, selective inheritance is easier to implement with concatenation than with delegation. However, our implementation of selective inheritance is already based on delegation. Thus, concatenation is not needed to provide all semantics of inheritance (although it might be beneficial for performance, but that is outside the scope of the Co-op/II prototype).

ORDERED HORIZONTAL COMBINATION For ordered horizontal combination, it should be possible to express an order between multiple parents of the same class. Using constraints to express such order might seem a nice solution. However, constraints are not expressive enough. Constraints are designed to express relations among different composition operators, not within the same composition operator. We could improve the expressiveness of constraints to allow expressing relations within a composition operator too, but we have not encountered any other situation that would benefit from this addition.

Another possibility for ordering the parents could be assigning an ordinal to every parent. Then, we could give control to a specific parent explicitly. This could be done by using an annotation, for example `@Parent(ordinal)`. Then, when the default binding fails, the annotation `@Parent(0)` would be attached to the message. If parent number zero is unable to handle the message, it rewrites the annotation to `@Parent(1)`. This process continues until a parent handles the message or no parent is left and the message send fails. This algorithm should be easy to implement with the current definition of Co-op. However, numbers are not implemented in the prototype, therefore it will be hard to provide an implementation that can be tested in the Co-op/II prototype.

SELECTIVE VALUE INHERITANCE Selective value inheritance is not implemented, but should not be hard to add. When using object-based inheritance, our library provides the choice between attribute and value inheritance. We also provide an implementation for selective attribute inheritance. Combining these two implementations would provide selective value inheritance.

4.5 RELATED WORK

Ostermann and Mezini [73] argue, fully in line with our reasoning that “[...] often non-standard composition semantics is needed, with a mixture of properties, which is not as such provided by any of the standard techniques.” To address this, they propose a small design space of properties of composition languages, which include *Overriding* of members, *Transparent redirection* of access to pseudovariables, and *Acquisition*, or transparent forwarding of access. This design space specifically covers the range of compositions from object aggregation to inheritance, but is unable to express other types of compositions, such as predicate dispatch, aspects, and so forth. A key technique proposed in their work are *Compound References*, which are exploited to express various alternative visibility and sharing

styles for data fields. Our approach differs among others in the ability to express crosscutting abstractions, and the ability to influence field access (dynamically) based on predicates.

Open classes [26], later called inter-type declarations in AspectJ [106], allow for flexibly extending classes with additional fields, expressed separately from the original class definitions. This allows for application-specific extension of classes with additional fields. Using advice on field read or write join points, it is also possible in aspect-oriented languages or frameworks to implement conceptual modifiers for field accesses. Examples are adding persistence³ or defining access permissions in the style of multilevel security for fields [82]. However, there is no notion of generalizing such extensions to an application-independent composition operator.

On another note, aspect-oriented languages such as AspectJ do not only allow for influencing field access and adding fields to classes; they also introduce the new mechanism of implicit object instantiation that is controlled by so-called *aspect instantiation policies*. Such policies declare when new aspect instances are shared between different (implicit) invocations and when a new instance has to be created. Compose* [84] extends this concept and supports instantiation policies per field in an aspect instance. In Co-op, it should be possible to realize all these mechanisms uniformly as composition operators.

Bracha and Lindstrom [14] discuss that the class construct (in languages with inheritance) has many different roles (nine, when ignoring type-related issues). They propose to adopt a very simple model of classes, which can then be enhanced by applying operators over modules, to express a wide range of inheritance semantics. As such it has similar aims as Co-op, but focuses only on inheritance-like semantics (e.g.: it is not able to express aspects), and it differs in its mechanism which is purely static; JIGSAW is a module manipulation language, whereas our composition operators express various composition semantics through first-class modules (expressed in the same language).

Reflex [96] is a reflection-based kernel for AOP languages. It also supports *structural aspects*, which may involve addition (and possibly modification) of members. Tanter and Fabry [97] discuss structural aspects, focussing on the detection of interactions between multiple, additive, and structural aspect expressions.

Reflective languages and systems in general offer low-level constructs for influencing, among others, the message dispatch process. However, for application programmers, they lack the abstractions provided by Co-op, such as bindings and constraints that enable the structured and composable expression of reusable composition operators. Nevertheless, fully reflective languages can be a suitable means for implementing languages such as Co-op, or for implementing a meta-object protocol that offers comparable abstractions.

Loker, Te Brinke, and Bockisch [P.7] describe a model for possible instantiation policies of aspects instances. This model describes when state is shared, which can be, for example, singleton, per object, or even per control flow. Even though their work focuses on aspect-oriented programming, such instantiation policies could be supported by Co-op. Instantiation policies are orthogonal to the data access described in this chapter, because this chapter only focusses on which state is available, not when it is instantiated.

³ See the SourceForge project “Java Persistence Aspect” at <http://sourceforge.net/projects/jpa/>.

The EventReactor language [64] is based on the Event Composition Model (ECM), which is an instantiation of the Co-op approach. The following list maps the elements from the ECM [64, chapter 7] to Co-op terminology:

EVENTS *Message sends*, which are generated in Co-op both by statements and by bindings, can generate new messages based on a selection (e.g.: grouping) of other messages.

EVENT CALLS *Calls* are always message sends, so all calls—also JNI calls—are effectively event calls.

REACTIVE PART Co-op has no explicit distinction between the reactive and non-reactive part. This means that all abstractions are also applicable to the reactive part.

CONTROL CALLS Providing information as arguments is possible for all calls, so also for control calls.

ARCHITECTURAL CONSTRAINTS *Constraints* are also provided by Co-op.

However, our realization—the Co-op language—has a different goal than the EventReactor language. Whereas we integrate all concepts in a single coherent language, EventReactor provides a separate language for managing events. The reason is that EventReactor assumes that programmers will implement the base program in various mainstream programming languages, but such mixture of languages increases complexity [33] and, therefore, we prefer a single integrated language.

5

TAILORING CONTROL FLOW

This chapter outlines how control flow tailoring can be added to Co-op, based on the publications “Reuse of Continuation-based Control-flow Abstractions” [P.3], presented at the 2nd Workshop on Free Composition, and “An Implementation Mechanism for Tailorable Exceptional Flow” [P.5], presented at the 5th International Workshop on Exception Handling.

In the previous two chapters, we have shown that Co-op provides tailorable composition mechanisms by influencing object behavior and data access. However, to modularize code, it should be possible to separate various control-flow related concerns, such as the expected and exceptional behavior, or the common flow and flow optimized for specific situations. Therefore, we want to allow tailorable control-flow mechanisms within Co-op.

The basics of control flow are method invocations, data access or expression evaluation. We are particularly interested in composition of control flows that are more complex than a simple request-reply model of method invocation. Examples of such complex flows are exception handling, co-routines, or around advice with proceed in aspect-oriented programming, which do not specifically address the receiver of a message send.

An example of tailoring control flow is introducing new kinds of exception handling, such as conventional try-finally and try-with-resources (introduced in Java 7), within a language. Application programmers should be able to choose which kind of exception handling they want to use. They may also choose to use control-flow mechanisms or abstractions that are not native to the adopted programming language. Of course, this choice should be expressed in a programmer-friendly way. Thus, application programmers should be able to think only about the use of exceptions, without dealing with the implementation details. Such implementation details can be left to library implementers. To allow this separation between *implementing* and *using* control-flow mechanisms, these mechanisms should be implemented modularly.

Design patterns provide a way to express compositions indirectly. The *command pattern* is a design pattern that is used to implement the control flow of calling methods at a later moment in time. A parallel loop, for example, can be implemented through the command pattern as the parallel execution of a set of invocations. More advanced scheduling policies for invocations are also possible. For instance, method calls can be scheduled in bursts, such as for the Media Player (section 7.4).

However, as explained in section 3.1.2, expressing such compositions indirectly has disadvantages. Therefore, we want to express control-flow composition directly in our language. This allows using parallel loops with ease. Composition operators could be even more advanced than the standard design patterns. For example, the loop that downloads in bursts can use the knowledge about the number of iterations left—possibly together with the actual state of the network manager—to trigger fast dormancy. This way, the separation between functional and optimization concerns is improved by control-flow composition operators.


```

1 var out
2 try {
3   out = File.open("A file that does not exist") //
      Raises an exception
4   out.write("Everything went fine")
5 } finally {
6   if (out != null)
7     out.close()
8 }

```

Listing 2: Traditional try-finally

```

1 var out
2 tryFinally({
3   out = File.open("A file that does not exist") //
      Raises an exception
4   out.write("Everything went fine")
5 }, {
6   if (out != null)
7     out.close()
8 })

```

Listing 3: First-class tryFinally

5.1 IMPLEMENTING TAILORABLE CONTROL FLOW

We identified two requirements for implementing the semantics of control-flow mechanisms as first-class abstractions:

- being able to influence the execution order of statements;
- having a way to associate various scopes to blocks so as to identify which variables can be used in which context.

We explain both requirements in more detail in the following two subsections.

5.1.1 Influencing Execution Order

Most important for tailorable control flow is the ability to influence the execution order of statements. Listing 2 shows a try-finally statement in pseudo code, which we will use as an example. (Leaving out the catch block is a simplification for the purpose of our discussion.) The statement on line 3 raises an exception, so the next statement to be executed is not the sequentially following one (line 4), but the first one of the finally block (line 6).

To make the behavior of the try-finally statement tailorable, it should be implemented as a first-class abstraction, like all composition operators in Co-op. We can achieve this by implementing it as a function to which both the try and finally block are passed as closures. This first-class implementation of try-finally can be used as shown in listing 3, which is analogous to listing 2. The try-finally statement and its semantics are no longer part of the language definition: the alternative execution order followed when an exception is raised can be expressed in the language itself; special language constructs are no longer needed, which improves the flexibility of the language. In this example, the semantics of the try-finally statement should be expressed by the implementation of the function tryFinally.

5.1.2 Associating Scopes

In section 5.1.1, we saw that a try-finally statement contains two blocks: the try and finally blocks. Inside both blocks, variables defined before the try-finally statement can be used, but variables defined inside the try block are invisible to the finally block and vice versa.

```

1 try (
2   var out = File.open(...)
3 ) {
4   out.write("Everything went fine")
5 }

```

Listing 4: Java-like try-with-resources

```

1 tryWithResources({
2   var out = File.open(...)
3 }, {
4   out.write("Everything went fine")
5 })

```

Listing 5: First-class try-with-resources

However, when we take a look at the try-with-resources statement of Java 7 [46, chapter 14.20.3], scoping is defined differently. A try-with-resources statement allows a set of `AutoCloseable` resources to be given, which are closed automatically when exiting the try block. Listing 4 shows an example of try-with-resources, which has the same behavior as the example described in section 5.1.1: a file is opened (line 2), something is written to that file if opening succeeds (line 4), and finally the file is closed (implicitly when execution of the try-with-resources statement finishes).

For this try-with-resources statement, we can define a first-class implementation, which can be used as shown in listing 5. Here, line 2 represents the resources block and line 4 the try block. We see that the try block does not only use the variables defined before the try-with-resources statement, but also the variables defined within the resources block.

The try-with-resources scoping differs from that of try-finally, but scoping is not limited to these two variants. For example, if a finally or catch block is added to the try-with-resources statement, variables defined in the resources block could be either accessible or inaccessible from the newly added block. Thus, we see that multiple scoping choices exist, which should not be limited to a fixed set provided by the language. Therefore, we conclude that *scoping should be tailorable*.

When scoping is tailorable, library implementers should also have the means to express which block uses what scope. Therefore, *library implementers should be able to associate scopes explicitly*.

5.2 INFLUENCING EXECUTION ORDER

Continuations [93] are a programming language feature that bundles a set of instructions together with an execution state into a first-class entity. They are referred to as a generalization of *goto*. In this chapter, we investigate the hypothesis that continuations may be a suitable atomic core principle that is able to express all, or many, of the well-known variations in handling control flow. Our eventual interest is in adopting continuations as a part of the generic composition model Co-op, able to express a wide variation of composition techniques through a single principled model dealing with control-flow composition *and* behavior composition.

There is a lot of literature that shows how existing language features for managing control flow can be implemented with continuations. We focus on investigating the reusability of continuation-based implementations of control-flow composition mechanisms, and discuss the continuation-based implementation of four variants of exception handling. The

Figure 17: Example of callcc

implementations are embodied in functions, for which Scala already offers reuse mechanisms. The main novelty of our work consists in the discussion of differences in the variant implementations. We will show that continuations are suitable to reusablely implement different variants of control-flow composition; but language features are required to compose and refine implementations of such mechanisms to avoid redundancy. Besides exception handling, which is presented in detail, we discuss a variety of additional control-flow composition mechanisms, of which several variants exist, and which can also be implemented with continuations.

5.2.1 Continuations

First-class continuations provide the ability to store an execution state (this is, e.g., the call stack together with an instruction pointer) and to continue execution in this state at a later point in the program [29]. The control operator *call-with-current-continuation* (callcc) invokes a function f and at the same time reifies the *current continuation* as a function object (see figure 17). The current continuation is the execution state at the time when the invocation of f is performed, where the instruction pointer refers to the instruction right after the invocation. This control operator implicitly passes the current continuation to f .

Continuation-passing style (CPS) is a programming style where no function call ever returns. Instead, the rest of the computation to be performed after the function finished must be passed as a continuation.

The continuation-passing style is often referred to as a mechanism suitable to implement many control-flow-related language concepts, for example co-routines, loops and exceptions. To our knowledge, existing publications only present the implementation of one variant of a specific mechanism. In contrast, our goal is to point out the commonalities in control-flow composition mechanisms and the potential for reuse in their continuation-based implementations.

We present implementations of various exception handling mechanisms and discuss their commonalities and differences. Co-op/III is designed to support all these mechanisms, but has only been implemented partly, as appendix C explains. All examples in this chapter are presented in Scala [71] and are tested with Scala’s semantics for continuations and context binding in continuations.¹ Scala is an object-oriented programming language with

¹ The showed examples are shortened, their full versions can be downloaded from <http://co-op.sf.net>.

```

1 def log(message) = {
2   var logger = null
3   try {
4     logger = new Logger("debug.log")
5     logger.log(message)
6   } catch { case e =>
7     println("Error: " + e.getMessage())
8
9     throw e
10  } finally {
11    if (logger != null) {
12      logger.close()
13    }
14  }
15 }
16 }
17 }
18 }

```

Listing 6: Using try catch finally in Scala

```

1 def log(message, ret, thro) = {
2   var logger = null
3   tryCatchFinally({ (ret, thro) =>
4     logger = new Logger("debug.log")
5     logger.log(message, ret, thro)
6   }, { (e, ret, thro) =>
7     callcc {
8       println("Error: " + e.getMessage(), _, thro)
9     }
10    thro (e)
11  }, { (ret, thro) =>
12    if (logger != null) {
13      callcc {
14        logger.close(_, thro)
15      }
16    }
17  }, ret, thro)
18 }

```

Listing 7: Using try catch finally with continuations

support for so-called delimited continuations [83]. This allows programmers to mix normal and CPS code, which we also use in our examples.

5.2.2 Exception Handling

To start with, consider the Scala code in listing 6 that uses Scala’s built-in exception handling mechanism. The operations within the `try{}` block may throw an exception—e.g., when the disk is full—which is in turn handled by the operations within the `catch{}` block; the operations within the `finally{}` block are always executed regardless whether the `try{}` block terminated normally or exceptionally.

5.2.3 Exception Handling with Continuations

Listing 7 shows the implementation of the same behavior with continuations. The main difference is that function calls are wrapped in a `callcc` block. As example, consider the call to `println` on lines 7 to 9; the current continuation is passed to `println` at the argument position denoted by ‘_’.

The structure of listing 7 is the same as that of listing 6: the call to `tryCatchFinally` has as arguments continuations representing the `try`, `catch`, and `finally` blocks. To emphasize this correspondence and to simplify our discussion, we also call these continuations “blocks.”

The implementation of the semantics of composing control flow in the presence of exceptions is embodied in the function `tryCatchFinally` (shown in listing 8) but also in the expected signature of the continuations passed to this function. The five arguments of `tryCatchFinally` represent the following:

```

1  def tryCatchFinally(dotry, docatch, dofinally, ret, thro) {
2      var finallyPlain = { () =>
3          dofinally(ret, thro)
4      }
5      var finallyRethrow = { e : Throwable =>
6          finallyPlain()
7          thro(e)
8      }
9      var finallyRereturn = { result =>
10         finallyPlain()
11         ret(result)
12     }
13     var catchPlain = { e : Throwable =>
14         docatch(e, finallyRereturn, finallyRethrow)
15         finallyPlain()
16     }
17     var tryPlain = {
18         dotry(finallyRereturn, catchPlain)
19         finallyPlain()
20     }
21     tryPlain
22 }

```

Listing 8: Defining try catch finally with continuations

1. The try block. This continuation must accept two arguments to be passed when the try block is executed:
 - a) The *return continuation* to be executed when the try block terminates normally.
 - b) The *throw continuation* to be invoked when the try block throws an exception.
2. The catch block. This continuation is called if an exception is thrown in the try block; it accepts the following three arguments:
 - a) The raised exception.
 - b,c) The return and throw continuations, which have the same role as the arguments of the try block.
3. The finally block. This continuation is called when the catch block finishes or when the try block finishes normally. It takes the same arguments as the try block.
4. The continuation to be executed when the try-catch-finally block terminates normally.
5. The continuation to be invoked when an exception thrown in the catch or finally blocks must be passed on to an enclosing exception handler.

In short, the implementation of `tryCatchFinally` works as follows. When we look at listing 7, the *try*, *catch* and *finally* blocks all terminate in one of three ways. They either (a) return from their defining method (i.e.: call the continuation `ret`), in this case log, (b) throw

(a) Control flow of *return* (b) Control flow of *throw*(c) Control flow of *fall off***Figure 18:** Using try catch finally with continuations

```

1 List(
2   (classOf[FileNotFoundException], // Exception type
3     (e, ret, thro) => { // Handler
4       callcc {
5         println("Invalid log filename: " + e.getMessage(), _, thro)
6       }
7       thro (e)
8     }
9   ),
10  (classOf[IOException], // Exception type
11    (e, ret, thro) => { // Handler
12      callcc {
13        println("Error while logging: " + e.getMessage(), _, thro)
14      }
15      thro (e)
16    }
17  )
18 )

```

Listing 9: Exceptions with multiple catch blocks

an exception (i.e.: call the continuation `thro`), or (c) ‘fall off’ (finish without performing any of the former two behaviors). The actual behavior of these three situations is defined in listing 8 by the continuations passed to the blocks, and illustrated in figure 18. (Not all statements in figure 18 are reachable, but are included to show which control flow would be the result of their execution. The gray blocks visualize that in practice there would be more code than shown.)

When the *try* block falls off, only the continuation `finallyPlain` is executed. When the *try* or *catch* blocks return, they invoke `finallyRereturn`, which executes the continuation `finallyPlain` before actually returning using `ret`. Since `ret` (line 11) never returns, `dofinally` is executed only once (line 19 is never reached in this case). When the *try* block throws an exception, it invokes the continuation `catchPlain`. When the *catch* block throws an exception, it invokes `finallyRethrow`, which executes the `finallyPlain` continuation before throwing the exception using `thro`. This implementation is inspired by an example presented by Might.²

5.2.4 Multi-Catch Exception Handling

From languages such as Java, developers are used to provide multiple catch blocks with a declarative definition of which type is handled by each of them. Listing 9 shows a modified catch block that uses this ability. It consists of a list of pairs specifying for which exception types (first element) the handler (second element) should be triggered.

Listing 10 shows a modified version of our try-catch-finally implementation that allows using the catch blocks presented in listing 9. A box encloses the code that differs from the

² See his blog post at <http://matt.might.net/articles/implementing-exceptions/>.

```

1  def tryCatchFinally(dotry, handlers, dofinally, ret, thro) {
2      var finallyPlain = { () =>
3          dofinally(ret, thro)
4      }
5      var finallyRethrow = { e : Throwable =>
6          finallyPlain()
7          thro(e)
8      }
9      var finallyRereturn = { result =>
10         finallyPlain()
11         ret(result)
12     }
13     var catchPlain = { e : Throwable =>
14         handlers.find(h => h._1.isInstanceOf(e)) match {
15             case Some((_, docatch)) =>
16                 docatch(e, finallyRereturn, finallyRethrow)
17                 finallyPlain()
18             case None =>
19                 finallyPlain()
20                 thro(e) // No handler, so throw on
21         }
22     }
23     var tryPlain = {
24         dotry(finallyRereturn, catchPlain)
25         finallyPlain()
26     }
27     tryPlain
28 }

```

Listing 10: Defining try catch finally, allowing multiple catch blocks

original version. The modification changes the handling of the exceptional case only. The `catchPlain` continuation invokes the first handler that can handle the thrown exception (i.e.: the thrown exception is assignment compatible with the type specified by the handler).

5.2.5 Retrying and Resuming Exceptions

Two more variants of exception handling are retrying [68] and resuming exceptions. With retrying exceptions, a catch block can re-execute the try block. For example when the disk is full, an I/O exception will be thrown. The catch block could show a dialog asking the user to free disk space and retry writing to the file afterward. With resuming exceptions, a catch block can resume the execution after the instruction throwing the exception. As example consider an I/O exception while writing to a file that is not vital; a catch block may try to recover from that, but may also decide to ignore this exception and skip the I/O operation.


```

1 { (e, ret, thro, retry) =>
2   callcc {
3     println("Error while logging: " + e.getMessage(), _, thro)
4   }
5   ... // Allow user to fix cause of failure, e.g. by freeing disk space
6   retry()
7 }

```

Listing 11: Catch block using retrying exceptions

```

1 { (e, ret, thro, ignore) =>
2   callcc {
3     println("Error while logging: " + e.getMessage(), _, thro)
4   }
5   ignore()
6 }

```

Listing 12: Catch block using resuming exceptions

Listing 11 shows the use of retrying exceptions, of which an implementation is given in listing 13. Again, boxes enclose the differences compared to the original version shown in listing 8, which lie only in the signatures of catch and throw.

Listing 12 shows the use of resuming exceptions. The try-catch-finally implementation for resuming exceptions is equivalent to that for retrying exceptions (shown in listing 13). The difference is their use: for retrying exceptions, the second argument of the throw continuation represents the start of the try block of the failing operation, whereas with resumable exceptions, it represents the rest of the try block's computation after the failing instruction.

As variant, resuming exceptions can be augmented by registering an exception handler higher up the call chain, which, e.g., ignores all I/O exceptions. This way, failing I/O will be ignored whenever it is not handled by any other exception handler. Another variant is multi-catch exception handling (shown in section 5.2.4) where the most specific handler is executed instead of the first matching.

5.2.6 Additional Possibilities of Continuations

In this section we presented implementations of four kinds of exception handling³ and showed that more variants exist. Additional examples of control-flow composition mechanisms that can be modeled with continuations are

- control statements, e.g.: for, while⁴, do, if, and switch,
- coroutines and generators⁴,
- concurrency [83],

³ Three of these types of exception handling are described at <http://c2.com/cgi/wiki?AbortRetryIgnore>.

```

1  def tryCatchFinally(dotry, docatch, dofinally, ret, thro) {
2    var finallyPlain = { () =>
3      dofinally(ret, thro)
4    }
5    var finallyRethrow = { (e : Throwable, retry) =>
6      finallyPlain()
7      thro(e, retry)
8    }
9    var finallyRereturn = { result =>
10     finallyPlain()
11     ret(result)
12   }
13   var catchPlain = { (e : Throwable, retry) =>
14     docatch(e, finallyRereturn, finallyRethrow, retry)
15     finallyPlain()
16   }
17   var tryPlain = {
18     dotry(finallyRereturn, catchPlain)
19     finallyPlain()
20   }
21   tryPlain
22 }

```

Listing 13: Defining try catch finally, allowing catch blocks to retry

- actors [83],
- reactive programming [83],
- asynchronous I/O and futures [83],
- monads [40, 83],
- lazy evaluation [101],
- iterators, selection and projection,
- around advice including proceed [35].

For most of these composition mechanisms, a continuation-based implementation is already given by Might⁴, Rompf, Maier, and Odersky [83], Wang [101], Filinski [40], or Endoh, Masuhara, and Yonezawa [35]. However, the similarities between the different approaches is not explored. We expect that different control structures will be overlapping in similar ways as exception handling, and also have variants that are not listed here. Therefore, we expect that these composition mechanisms have similar reuse opportunities as we showed for exception handling.

⁴ See the blog posts of Matt Might at <http://matt.might.net/articles/>, section Continuations.

Hannema [T.5] has developed a visual specification language for control structures. He has shown that his visual language supports control structures such as subroutine calls, for loops, try-catch statements, and reentrant generators. The visual models can be automatically translated to a continuation-based implementation in Scheme.

5.2.7 Weaknesses of Control-Flow Composition with Continuations

As shown in this section, different variants of control-flow composition mechanisms share the largest part of their implementation. At the same time, we saw that first-class continuations are suitable for implementing variants of control-flow composition. By implementing the semantics of such mechanisms in terms of a function, it is possible to provide the implementation as library and developers can choose to use any of the variants, even multiple variants in the same project. Since there are well-established mechanisms for reusing function definitions, a continuation-based implementation fulfills our reusability requirement.

To make this approach actually feasible, we see two points that need improvement. First, reusing control-flow composition mechanisms as a whole is already possible as outlined in this section. However, we recognized that there is large overlap between implementations of different control-flow composition variants. Therefore, new language features are required to compose and refine implementations of such mechanisms so as to avoid redundancy. It should be possible to implement the variants, such as multi-catch-block and retrying or resuming exceptions, as extensions of the original try-catch-finally implementation without any code redundancy. Ideally, it should also be possible to compose several variants easily to, e.g., define exceptions which can both be retried and resumed.

Second, the continuation-passing style puts a burden on the application developer as the code becomes more complicated than in a programming style where functions always return. We would like to hide the complexity of programming in continuation-passing style while still being able to use its ability to modularize the implementation of control-flow composition mechanisms.

5.2.8 Conclusion

The continuation-passing style is a powerful mechanism for influencing execution order. However, programming in this style is—in general—less readable than programming in a direct style. Therefore, application programmers should not have to program in the full continuation-passing style. However, library implementers should be able to influence the execution order.

For example, whether or not the current continuation is passed to a function or continuation implementing a composition mechanism, determines its semantics. This has been shown by the example of retrying and resuming exceptions, which only differ in which continuation is passed to the throwing continuation. Thus, a function should declare whether it is called with the current continuation or not, rather than requiring that invocations are in a `callcc` block.

For the try-finally statement, library implementers can define the concept `throw` as a continuation. Once defined, such a continuation is available implicitly to application pro-

```

1  tryWithResources(
2  ← [ { var out = File.open(...) }
3
4  ← [ { out.write("Everything went fine") }
5  )

```

Listing 14: Default scoping

```

1  tryWithResources(
2  ← [ { var out = File.open(...) }
3
4  [ { out.write("Everything went fine") }
5  )

```

Listing 15: Desired scoping

grammers. Thus, when using the try-finally control-flow mechanism, application programmers can invoke `throw()` to raise an exception, without writing all code in the continuation passing style.

5.3 ASSOCIATING SCOPES

Most programming languages structure scopes as a tree or forest, in which every scope has at most a single enclosing scope. In general, these scopes are lexically nested. Listing 14 shows an example of nesting; the boxes depict the scopes and the arrows the nesting (they point towards the enclosing scope). Such a tree structure is sufficiently powerful for modeling the scopes of a wide range of control-flow mechanisms. However, as shown in section 5.1.2 and in listing 15, depending on the control-flow mechanism, the choice of enclosing scope might vary. Therefore, we propose using first-class scopes. These scopes become accessible properties of continuations, of which library implementers can alter the enclosing scope. In this way, scopes can be organized in any desired tree structure.

Theoretically, it is possible to create a cyclic graph if enclosing scopes can be freely re-assigned. However, cycles can lead to infinite recursion during variable lookup, when referring to an undefined variable. Such infinite recursion can be avoided, but since we have not seen any use cases of cyclic scopes, implementations might disallow cycles in the scope graph. In that case, the scope graph will be a directed acyclic graph, which—together with the requirement that every scope has at most one enclosing scope—implies that the scope graph is a forest.

5.3.1 Scope Associations

Section 5.1.2 explained the purpose of scopes when *using* a control-flow mechanism. Now, we will show an example of their purpose while *defining* a control-flow mechanism. This example is the try-finally statement, which was explained in section 5.1.1.

Listing 16 shows an envisioned implementation of `tryFinally`. Lines 2 to 11 specify how scopes and continuations are associated, termed *associations*, and lines 13 and 14 specify the actual function body.

Associations refer to continuations that are either passed as arguments (try and finally, in this example) or available implicitly (return and throw, in this example). Associations can both call these continuations and access their properties, which are either scopes or continuations. Common scopes are: the scope created by a continuation (scope), the enclosing

```

1 method tryFinally(var try, var finally) [
2     try.enclosingScope = try.definingScope
3     finally.enclosingScope = finally.definingScope
4     try.return = { value →
5         finally()
6         return(value)
7     }
8     try.throw = { exception →
9         finally()
10        throw(exception)
11    }
12 ] {
13     try()
14     finally()
15 }

```

Listing 16: Definition of tryFinally

```

1 method tryWithResources(var resources, var try) [
2     resources.enclosingScope = resources.definingScope
3     try.enclosingScope = resources.scope
4     try.return = { value →
5         for (resource defined in resources)
6             resource.close()
7         return(value)
8     }
9     try.throw = { exception →
10        for (resource defined in resources)
11            resource.close()
12        throw(exception)
13    }
14 ] {
15     resources()
16     try()
17     for (resource defined in resources)
18         resource.close()
19 }

```

Listing 17: Definition of tryWithResources

scope used during execution of a continuation (`enclosingScope`), and the scope in which the continuation is defined (`definingScope`). Common continuations are `return` and `throw`, which represent the behavior of returning from a method and raising an exception.

Looking at listing 16, we see that both the `try` and `finally` blocks close over the scope in which they are defined (lines 2 and 3). Further, inside the `try` block, `return` and `throw` cause `finally` to be executed before actually returning or throwing, respectively. For example, lines 4 to 7 specify the continuation `return`, which receives the value to be returned (line 4) and executes `finally` (line 5) before actually returning this value (line 6).

Our proposed implementation of `tryWithResources` uses different scoping. When we look at line 3 of listing 17, we see that the `try` block closes over the scope used by the `resources` block. Further, instead of calling `finally`, it closes all defined resources (lines 17 and 18). The precise way of accessing all defined variables is future work.

5.3.2 Influencing Scopes and Execution Order

With the use of continuations, a method can act as a control-flow operator. Using such a control-flow operator is basically the same as calling a method without a receiver. Together with the ability to influence scoping, e.g., loop constructs with different semantics can be defined, such as evaluating the loop condition before, after or even during the loop [28].

Since we do not require the sender to explicitly write calls in continuation-passing style, such loop constructs can be used by programmers in a conventional way. Library implementers who define such loops can, however, use the continuation-passing style for implementing the semantics of the loops.

5.4 RELATED WORK

Publications on implementing control-flow abstractions with continuations have been discussed throughout this chapter. Additional documentation [3, 61] presents approaches for compiling control-flow statements to continuation-passing style. To the best of our knowledge, the *comparison* of control-flow abstractions based on a common implementation technique, however, is missing in literature. Therefore, we compare the implementation and use of various kinds of exception handling.

Metaspin [16] uses continuations for implementing around advices, but does not express these as first-class entities that can be accessed by the programmer.

Wakita [100] uses two variants of first-class continuations: method and message continuations. Method continuations are first class continuations similar to the ones we use, message continuations are used to create first-class messages. However, continuations are only discussed from an implementation perspective; there is no discussion of their usage by application programmers.

In Pharo [9, chapter 13]—a Smalltalk environment—various kinds of exceptions are implemented uniformly. However, their implementation does not use a generic language mechanism suitable for implementing other abstractions, but relies on VM support that is specific to exception handling.

To overcome the complexity brought by CPS, C# 5 introduces the keywords *async* and *await* to achieve asynchronous computations.⁵ For ECMAScript extended with generators, which adds the keyword *yield* to the language, the same is achieved as library functions.⁶

Several unified implementation techniques are discussed in literature, which, however, are limited to modeling a request-reply-style of composition rather than arbitrary control flow composition as presented in this chapter. As examples consider the work on predicate dispatching by Ernst et al. [36] or on associative lookup by Piumarta [79].

Neron et al. [70] describes a theoretical model for name resolution, which includes both enclosing and importing scopes. In their approach, first a scope graph is created, on which then resolution calculus is applied. Our approach is complementary, because we provide a tailorable way to create the scope graph. Specifying our name resolution with the resolution calculus of Neron et al. and augmenting it with imports would be an interesting future direction for further formalization of our approach.

⁵ See Eric Lippert's blog posts at <http://blogs.msdn.com/b/ericlippert/archive/tags/async/>.

⁶ See the ECMAScript wiki at http://wiki.ecmascript.org/doku.php?id=strawman:async_functions.

Part II

RESOURCE UTILIZATION MODELS

This chapter introduces our notation for resource-aware software, which was published in “Towards Modular Resource-Aware Applications” [P.4], presented at the 3rd International Workshop on Variability & Composition, and “A Tool-Supported Approach for Modular Design of Energy-Aware Software” [P.11], presented during the 29th Annual ACM Symposium on Applied Computing.

A software engineering challenge is managing the complexity of software [86]. We are building increasingly large software systems. Such systems encompass a substantial amount of *inherent complexity*; partly in the problem domain and partly in the solution domain. However, the *realization* of these systems also introduces a large amount of *accidental complexity* [17], while going from the conceptual solution to a realization model. The limited ability of realization models to accurately represent the concepts and their interdependencies in a conceptual solution is the main cause of accidental complexity. As a result, the complexity of realizations is typically substantially larger than the complexity of the conceptual solution. Thus, to optimize the resource utilization of software without introducing accidental complexity, we need a notation that supports modeling the resource utilization of components.

Existing modeling languages, such as UML, do not offer generic support for modeling the resource utilization of components. However, dedicated UML Profiles [72] allow the description of *specific* sets of common software and hardware resources.

Research on explicitly modeling resource utilization exists [2, 89], but these approaches do not take software modularity into consideration. Also, some middleware is aware of *quality of service* (QoS) [77] and resource consumption can also be considered a QoS. However, the optimization of QoS in middleware assumes that all optimizations are handled through a single middleware layer.

We provide a component model that can specify more advanced resource utilization than QoS and helps to define resource utilization in a modular way.

6.1 REQUIREMENTS FOR RESOURCE-AWARE SOFTWARE

Our goal is to facilitate the structured development and maintenance of resource-aware software systems. A key technique for achieving this is to separate concerns and implement them in separate modules. These separate models must then be combined or *composed* into an integrated system. We can observe several modeling requirements when designing resource-aware systems according to this approach; namely, the need to

PERFORM OPTIMIZATIONS IN VARIOUS LOCATIONS Depending on the particular system and optimization techniques, optimizations may involve and affect multiple software mod-

ules within the system. There is no single optimization technique that is generally applicable. Optimizations may cross-cut the system and affect or control the behavior or state of other modules.

SEPARATE OPTIMIZATION AND FUNCTIONALITY To manage complexity and evolution, optimization and functionality should be specified separately, even though these are—to varying degrees—inter-dependent. Such modular specification should be composable into an integrated system.

SEPARATE FUNCTIONAL AND RESOURCE-USAGE INTERFACES Resources must be managed and represented explicitly (so they can be optimized) and must be independent of functional modules. Therefore, we must provide explicit interfaces for both functionality *and* resource usage—both required and provided resources.

KNOW THE RESOURCE BEHAVIOR OF OTHER MODULES Advanced optimizations do not only rely on measuring the state of the system (such as resource usage), but need information about the ongoing and expected activities from the involved modules. Thus, modules must provide information about their resource usage *behavior* to other modules, to facilitate composing these modules.

The above list emphasizes the engineering requirements for separation of concerns during modeling. Separate concerns can then be implemented in independent modules (we deliberately disregard technology specifics here). However, these modules can in some cases be tightly related, and always need to be composed into a single integrated system.

6.2 A NOTATION FOR RESOURCE-AWARE SOFTWARE

Traditionally, a component is considered to be a unit of development and deployment, with explicit interfaces specifying the services that it provides to and the services that it requires from its environment [94, chapter 5]. We extend components with explicit interfaces specifying the *resources* that it provides to and requires from its environment. Besides its implementation, each component encapsulates a so-called *resource-utilization model* that expresses the relation between the component services and resources.

Figure 19 represents our notation of components, where the boxes *Resource-Utilization Model (RUM)* and *Component implementation* are placeholders, of which the RUM will be explained in the next section and the implementation could be provided in any programming language. In the following, we explain all other parts in more detail (starting at the top-left and continuing clock-wise):

PROVIDED RESOURCES This is a separate description for all resources a component provides, including the type of these resources, and—as appropriate—static constraints on the availability of these resources.

PROVIDED SERVICES The functional behavior of a component is specified as separately described services. The key issue here is that typically, each service will consume resources, and the component specification describes which resources these are, with—

Figure 19: Notation for resource-aware components

as appropriate—static constraints on the consumption of these resources. The resource consumption of an individual service invocation may be influenced by the values of its parameters.

REQUIRED SERVICES Similarly, a component may require certain services from other components, to fulfill its duties. The required services may also specify constraints with respect to resource usage that provided services must adhere to (such as a maximum amount of consumed power).

REQUIRED RESOURCES A component may specify that it requires a certain amount of resources; for example, because the component encapsulates (hardware) behavior that requires certain resources, or because the component will provide these resources—often with some restrictions or policy—to other components.

RESOURCE-UTILIZATION MODEL (RUM) Resource-aware components may declare their resource behavior, i.e., the dynamic relation between resources and services. For example, they may specify that using a certain service of a component increases the availability of a provided resource or puts a component in a state where it consumes more of a required resource. The RUM specifies the impact on resources in detail, e.g.: the degree to which the availability or consumption of the resource changes, the end-condition for staying at this level, the availability of resources and services in different situations, and so on.

6.3 RESOURCE-UTILIZATION MODEL (RUM)

In contrast to the services and resources, which are basically identifiers, the resource utilization model (RUM) of the component is more complex: it is a state machine that represents the relations between all services and resources provided and required by the component. Since a RUM is too complex to be represented as an identifier annotated to a port, it is represented as a box inside the component and exposed through an octagonal port.

In software engineering, a key strategy for dealing with change and managing complexity is “divide and conquer”: achieve *separation of concerns* [31] by dividing a solution into

Figure 20: Energy state chart

building blocks and delivering working systems by expressing proper compositions of these building blocks. RUMs are examples of such building blocks, which allow the separation of optimization from the functional concern.

As it is shown in section 6.4, we consider state machines a suitable notation for representing resource-utilization models, because (a) they are declarative, (b) their transitions can model invocations on the services of the component, (c) they can also model the internal behavior of a component and its relation with the services that the component provides to and requires from its environment, and (d) they can conveniently be extended with resource-utilization annotations on states as well as state transitions [8]. Such information can be used by resource optimizers to make more adequate decisions, based on how various components provide and consume resources.

An example of a state machine with energy annotations is shown in figure 20. States are annotated with their power usage (energy consumed per second), and transitions with the energy needed for taking that transition. These annotations can be constants, such as 1 J/s, or formulas containing the variables defined in the state machine, such as $(a - 10)$ J/s.

With respect to optimizing the resource consumption of a system, components can have one or more of the following responsibilities: providing, consuming, or controlling resources. Pure providers are, for example, drivers that encapsulate hardware resources such as power, memory, or bandwidth. Pure consumers are, for example, application components that consume one or more resources. Controller components can be optimizers that require resources from the resource providers and provide them to consumers after applying certain resource-optimization algorithms. Our notation can express these three kinds of components. For example, (a) a resource provider does not have any resource consumption interface, (b) an application component does not have any resource provision interface, and (c) a controller component has both kinds of interfaces.

6.4 EXAMPLE APPLICATIONS

To illustrate the use and expressiveness of our notation, we have modeled two architectures of energy-aware software in our notation. Both example applications are adapted from publications that discuss the energy consumption of the software, without taking modularity into account. We show how the architecture of these applications looks in our notation, to show that we can achieve a modular design.

6.4.1 The Hiker's Buddy Application

As first example we discuss the Hiker's Buddy application [34], which receives information sent by a Global Positioning System (GPS) receiver and plots the hiker's current location on a topographical map that is retrieved by searching a database of map segments. This application is necessarily mobile and battery-powered. Therefore, the software must be made energy-aware so that the battery life-time can be maximized.

ARCHITECTURE The main components of Hiker's Buddy are depicted in figure 21 (the dashed lines group the states as provided by Ellis [34, figure 1]). The application *Hiker's Buddy* uses an *LCD* to show maps to the user. These maps are loaded from a *Database*. In order to know the current position of the user, a *GPS* is used. The *GPS* component encapsulates both acquiring satellite information and providing this information to the system—which is done over a serial line. The *Optimizer* can be used to optimize the *GPS* utilization, which will be explained in the next paragraphs. The *Power Supply* delivers battery power to the components.

RESOURCE OPTIMIZATION Polling the GPS for location information is a costly power state. Its total energy expenditure depends on both power consumption of the state and the time spent in this state. These are influenced by the polling frequency of GPS data: the faster polling occurs, the faster Hiker's Buddy can discover the availability of a good GPS fix, thus reducing the time spent in a costly power state; but polling less frequently allows the processor to spend more time in idle mode, thus lowering the overall power cost of this loop. If that does not significantly affect the total time for the loop, energy use is reduced. An adaptive polling frequency is usually a better policy for energy use than any fixed polling interval, e.g., by increasing the polling frequency when a GPS fix is getting close.

RESOURCE-UTILIZATION MODELS A high-level model of the power states for Hiker's Buddy—which is adopted from Ellis [34]—is shown inside the *Hiker's Buddy* component in figure 21. For readability, the states are divided into two parts by the vertical dashed line: (a) the states to the left of the line capture the device requirements remaining when another application is in the foreground, and (b) the states to the right of the line represent Hiker's Buddy as the foreground application. Every state is annotated with the approximate power consumed by the device while it is in this state. The state *GPS SL* is actively reading data from the GPS while Hiker's Buddy is in the foreground; it is the most power consuming state. The *main eventloop* polls the GPS for the current location by switching to the *GPS SL* state at regular intervals.

The *GPS* component does not only offer the current location, but also information about the acquired satellites. Using this information, *Optimizer* can estimate how close a GPS fix is and—together with the information provided by the RUM of *Hiker's Buddy*—optimize the polling interval.

The power utilization shown in this example was acquired by Ellis, measuring the power consumption of the device as a whole. Thus, we only have enough information to present a RUM for the Hiker's Buddy component, which includes the power consumption of all other components. While this is sufficient to perform the desired optimization, it hinders separate

Figure 21: A component diagram for Hiker's Buddy

maintainability of the different components. If, for example, we want to replace the *Database* or *GPS* component, we cannot see how this would influence the resource-utilization model. In order to do so, the whole RUM must be updated. Therefore, we claim that *every component should have its own RUM*.

6.4.2 Smart Phone Network Traffic Reduction

The second and running example applies our notation to model the solution proposed by Qian et al. [81] for optimizing the power consumption of smartphones. Because smartphones are also mobile and battery-powered, and transmitting data over the 3G radio network consumes power, reducing network traffic can maximize their battery life-time.

ARCHITECTURE The high-level architecture of the smartphone device shown in figure 22 contains four components. The component *Application* represents mobile applications that require services from the 3G network. In our example, we focus on one application only, i.e., a media-player application, but in practice multiple applications could be executing simultaneously. The media-player mobile application provides three services: *play*, *pause*, and *stop*. The component requires the services *connect*, *disconnect*, *download*, and the resource *connection*. The component *Optimizer* improves the usage of connections by the applications, which consequently leads to the optimization of radio power consumption.

The component *Network Manager*—offered by the 3G network—provides the actual data transfer over the network, for which it requires the resource *radio power* provided by the component *Power Supply*.

RESOURCE OPTIMIZATION In cellular networks, the radio connection of phones to the 3G network consumes power. To efficiently utilize this limited power, the component *Network Manager* introduces a so-called inactivity timer for each phone and degrades or releases connections if a phone is inactive for the specified amount of time. Although this reduces the radio-power consumption and bandwidth usage, during the inactivity time, radio power is still being consumed to some extent. Therefore, power consumption is further improved by introducing the component *Optimizer* as mediator between *Application* and *Network Manager*. It explicitly commands *Network Manager* to release connections when they are no longer needed by *Application*.

RESOURCE-UTILIZATION MODELS The internal resource consumption of the component *Application* is depicted in figure 22 as a state machine. Here, the state *playing* is the start state. To buffer media content, in this state the resource *connection* is consumed by invoking the service *connect* followed by *download*. When the first m seconds of media content are buffered, the connection is released by invoking the service *disconnect*. The connection is again established after n seconds of the buffered content are played.

The invocation of the service *stop* causes a transition to the state *stopped* that does not need the resource *connection* anymore. Therefore, it invokes the service *disconnect*. The invocation of the service *pause* causes a transition to the state *paused* if the state chart is in

Figure 22: A component diagram for smartphone network performance

the state *playing*. In the state *paused* the resource *connection* is required until m seconds of media content are buffered.

The component *Optimizer* uses the algorithm proposed by Qian et al. to manage the connections for the applications. In short, when an application invokes the service *disconnect*, it also identifies the next time when it requires a connection. If all applications require the connection no sooner than x seconds (x is defined by the RUM of *Network Manager*), *Optimizer* invokes the service *disconnect* on the component *Network Manager*; otherwise, it keeps consuming resources. In this way, it keeps a balance between download latency and resource consumption.

The component *Network Manager* keeps a state machine to manage the bandwidth consumption of the phone. The available bandwidth and the power consumption vary per state. Here, the state *IDLE* is the default state, indicating that the phone has not established a connection, thus no bandwidth is allocated and no radio power is consumed. The state *CELL DCH* indicates that the phone is allocated dedicated transport channels in both downlink and uplink. The state *CELL FACH* indicates that a connection is established but there is no dedicated transport channel allocated to a phone. Instead, the phone can only transmit user data through shared low-speed channels. Consequently, this state consumes less radio power than the state *CELL DCH*.

To manage the connectivity and, thus, the resource consumption, *Network Manager* defines inactivity times and switches between states. For example, as shown in figure 22, after x seconds of idle time a state transition occurs from *CELL FACH* to *CELL DCH*, and after y seconds of idle time a transition occurs to *IDLE*.

6.4.3 Modularity of Case Studies

Our proposed notation has been designed to support the structured development and maintenance of software for resource-aware systems. We can observe how the case studies motivate the modeling requirements:

PERFORM OPTIMIZATIONS IN VARIOUS LOCATIONS There is no single optimization technique that is generally applicable; for example, in the *Hiker's Buddy*, there is a more centralized control, whereas the *Smartphone* optimization involves network-layer support, a centralized controller, and application-level functionality.

SEPARATE OPTIMIZATION AND FUNCTIONALITY In both case studies, optimization is implemented *outside* the various modules in the system—but it does require communication with those modules.

SEPARATE FUNCTIONAL AND RESOURCE-USAGE INTERFACES This is not strongly emphasized by the case studies, but derived from the need to manage and represent resources explicitly (so they can be optimized), and the need for independent (functional) modules.

KNOW THE RESOURCE BEHAVIOR OR PLANS OF OTHER MODULES For example, the optimizing *controller* for the *Smartphone* must know when the applications expect to reconnect.

6.5 FORMAL DEFINITION OF RUMS

To analyze (composed) software systems for showing correctness and predicting resource consumption, the meaning of RUMs must be formally defined. This section refines the general description of RUMs, given in section 6.3, to a more rigorous definition.

Modularity means that the implementation of a concern can be exchanged without touching other implementations. In our case, the resource optimization concern should be modular. Therefore, the RUM of a component must be an *abstraction* of its concrete implementation. This provides the ability to replace components with other implementations satisfying the abstraction, while maintaining the properties already proven on the composed system.

That RUMs are abstractions in turn means that a RUM only specifies implementation details that are relevant for reasoning about the resource utilization, and leaves out irrelevant details. In our design approach, we propose to determine *relevant* details by specifying so-called *key properties*, which need to be established for a system. In the following subsection, we elaborate on the nature of the key properties we express. Afterwards, we discuss what characteristics the RUM must have to support reasoning about models in the desired ways.

6.5.1 Identifying Key Properties

The key desirable properties, both those relating to the functionality of the system and those relating to resource consumption, can be derived from the requirements specification of the software. Properties of both kinds can be expressed in terms of the possible execution sequences of the system, and are commonly specified using a temporal logic notation [23, 65]. (Other notations can also be used and we do not discuss the precise notation here, but we assume that all properties are trace based.) Which key properties can be checked automatically depends on the formalism in which the properties and the RUM are expressed. To identify which formalism we need, we must first identify which kinds of properties we want to check. We classify the properties into two dimensions:

- Which execution sequences should be considered? Properties can specify that they hold (1) for *every* execution sequence of the system or (2) for *some* (i.e., at least one) execution sequences.
- Is the property (a) a *safety* property that should hold in every state of considered executions (and can be violated in a finite prefix of an execution sequence) or (b) a *liveness* property that ensures reaching desired states in the executions considered (and can only be violated in (usually infinite) execution sequences that do not reach the desired states).

The combinations of these two dimensions are enumerated in the following list, where Φ represents a state predicate.

1. For **every** execution sequence,
 - a) Φ always holds. (Safety)
 - b) Φ will eventually hold in the future. (Liveness)

2. For **some** execution sequence,
 - a) Φ always holds. (Safety)
 - b) Φ will eventually hold in the future. (Liveness)

For example, for a media player we can use the property “music is playing” in place of Φ :

- 1a. Throughout every execution of the media player, music always plays.
- 1b. Pressing the play button always results eventually in playing music.
- 2a. There are system execution sequences where music is always playing.
- 2b. For some execution sequence, pressing play will eventually result in music playing.

Instead of only specifying functional behavior, properties relating to the amount of resource consumption can also be specified, using state assertions of the following types:

- (i) Min/max. “The current resource consumption is within a certain range.”
- (ii) Total. “The total resource consumption is within a certain range.” (This is the integral over the current resource consumption.)

Most critically, we would like to use verification tools to help find optimal execution sequences for a given sequence of user requests and system responses, or for a finite family of such inputs:

- (iii) Optimization. “Provide a sequence for which the resource consumption is minimal.”

Examples of properties are:

- The media player always uses less than 1 J/s (1a, i)¹
- For every execution sequence, whenever a certain song is fully played, the energy consumed since the song was requested is less than 10 J (1a, ii)
- For some execution sequence, the media player reaches a state after fully playing a certain song, consuming less than 10 J since the song was begun (2b, ii)

Standard temporal logic and model checking do not allow directly answering an optimization request of type (iii). However, the nature of an unsuccessful model check—that it provides a counterexample when the property does not hold for the model—can be exploited. For example, when an assertion of the form “For every execution sequence, whenever three minutes of music have been played, the energy consumption is greater than 10 J” is *not* true of the system, an attempt to model check shows a counterexample where the energy consumption is *less* than 10 J. In this way, an optimized computation (e.g., which buffers network data and works in bursts) can be detected. Exploring this possibility for *finding* optimizations is left as future work. In this section, we will focus on analyzing systems and comparing possible optimizations of these systems.

¹ For clarity, we write energy consumption as Joules per second (J/s) instead of its equivalent Watts (W), because we express the total amount of consumed energy in Joules, not in Watt hours.

Figure 23: Example over and under-abstraction for a coffee machine

6.5.2 Modeling Resource Behavior

In component-based software development, service interfaces of a component are abstractions over the actual implementation—the behavior—of the component, and give guarantees about the functional requirements that are fulfilled by the component. Such guarantees facilitate using components without knowing their implementations, and enable flexibly changing the implementations of components, provided that the interfaces remain intact. Since we consider RUMs as part of the component interface, RUMs must also (a) provide abstraction over the behavior of the component, and (b) provide guarantees on the resource utilization of components. Therefore, properties 1a, 1b, i, and ii are most important, because 1a and 1b provide guarantees for the model and can be combined with i and ii.

Abstraction over Component Behavior

RUMs abstract over the behavior of a component, which we define as all possible execution sequences of that component. For RUMs, there are two kinds of abstraction to consider:

OVER-ABSTRACTION An over-abstraction (also called an overapproximation) specifies a superset of the possible execution sequences of a component. If we verify a property that holds for all execution sequences in the over-abstraction, then it also holds for the execution sequences in the actual concrete system. For example, an over-abstraction can be used to prove an upper bound of the maximum energy consumption of the actual system and a lower bound of the minimum energy consumption. Thus, over-abstractions provide guarantees of the component.

UNDER-ABSTRACTION An under-abstraction (also called an underapproximation) specifies a subset of the possible execution sequences of a component. If we verify a property for some execution sequence in the under-abstraction, then that computation is also in the actual system, and the property holds there too. For example, an under-abstraction can be used to find an execution sequence that gives a lower bound of the maximum energy consumption and an upper bound of the minimum energy consumption of the actual system. Thus, under-abstractions give possibilities of the component.

The coffee machine in figure 23a provides an example. Both specifications in figures 23b and 23c are abstractions of the coffee machine, because the specifications are smaller than

Figure 24: Relation between actual system and its abstractions

the actual system; they contain fewer states or fewer transitions, respectively. Figure 23b is an over-abstraction, because it has more possibilities than the actual machine: it can provide coffee for free. Figure 23c is an under-abstraction, because it has fewer possibilities than the actual machine: it cannot provide coffee (only tea can be served).

Figure 24 shows the relation between these kinds of abstractions and the actual behavior. At the left, we see which execution sequences are included in the respective abstractions. At the right, we see the bounds that can be guaranteed by analyzing the respective abstractions. For example, with over-abstraction we will never overestimate the minimum and never underestimate the maximum consumption and thus acquire a conservative worst-case result.

It would be possible to create abstractions that are neither over nor under-abstractions, but such abstractions provide none of the guarantees that are provided by either over or under-abstractions. Therefore, such abstractions are not useful in a formal context.

Guarantees of Component Resource Utilization

The RUM of a component must provide guarantees about resource-utilization requirements of that component and the resource utilization requirements are formulated as key properties that hold for all execution sequences. Checking that these requirements hold can only be performed on RUMs that are over-abstractions; it would not be possible if RUMs were expressed as under-abstractions. This means *to fulfill most of the typical requirements, a RUM must be an over-abstraction*.

6.5.3 Analyzing Resource Behavior and Selecting Optimizer

When a software system is designed following the guidelines defined in this section—i.e., using a component model that specifies RUMs as over-abstractions of the components' behavior—the resource behavior of the composed system can be analyzed. The resource behavior of the overall system is determined by the composition of all the separate RUMs.

Therefore, the designer can analyze the combined RUMs to understand the resource consumption of the system.

Because the resource utilization is specified for each component modularly, it is easily possible to analyze various compositions and especially to investigate the influence of alternative variants of single components. More specifically, it is possible to design multiple alternative optimizer components and use the analysis to identify which optimization results in the least resource consumption under various usage scenarios. Based on this analysis, designers can select a composition for the final system design using the most suitable among alternative optimizer components.

As explained in the previous two subsections, it is most relevant to express guarantees of the resource behavior of components. The same argumentation holds for analyses of the composed system. Therefore, it has to be noted that the combination of the individual RUMs is an over-abstraction of the composed system, just like each single RUM is an over-abstraction of a single component. Thus, again, every property that we can prove to hold for every execution sequence in the combined RUMs, also holds for every execution of the concrete system. This means that we can perform an analysis of the properties of type 1a, 1b, i and ii as specified in section 6.5.1, i.e., safety and liveness properties specifying ranges and totals of resource consumption for every execution.

This chapter describes a design process that can be used to model resource-aware applications according to the notation shown in the previous chapter, based on our publication “A Design Method For Modular Energy-Aware Software” [P.9], presented at the 28th Annual ACM Symposium on Applied Computing, and its accompanying technical report [P.8].

A modeling notation is, by itself, not sufficient to achieve modularity in the design of energy-aware software. Designers need guidance through the activities that must be performed to identify and modularly design (a) necessary components, (b) the models that must be prepared during each activity, and (c) the analysis that must be performed on the models.

This chapter describes a workflow for designing energy-aware software. The workflow helps designers identify the kinds of components that are typically needed to be taken into account, the energy-specific interfaces of these components, their interaction with one another and the analyses that must be performed on them. We also explain a concrete realization of this workflow by means of the UPPAAL tool [7], and illustrates the suitability of UPPAAL to analyze the design models. An application of this workflow to modularly design a real-life media-player software is illustrated.

Figure 25 is a UML activity diagram that depicts the activities that are performed in our workflow, along with the dependency among activities. The activities are shown as boxes in the figure and each activity results in a model. The arrows represent the order of activities; the activities between bars can be performed in any order. The diamonds represent points in our method where the models are evaluated and possibly are redesigned iteratively, e.g., by changing the decomposition, or adding details. The activities result in a set of modeled components, which are represented in the notation depicted in figure 19 on page 65.

7.1 OVERVIEW

At the top of figure 25 are the activities for identifying the software components. The design method distinguishes among three kinds of components: *functional*, *user* and *optimizer*. The functional components refer to both software and hardware components that form the target system; each component implements part of the functionality of the system and interacts with other software and hardware components to accomplish the overall functionality. Since there are already various guidelines for modularizing functional aspects of software [44, 74], we suggest to adopt an existing guideline to identify functional components.

The way in which a user interacts with the software plays a role in its energy consumption [47]. To be able to analyze the effectiveness of an optimization strategy, our workflow models the *user* as a component of the system in terms of usage scenarios.

Figure 25: Workflow for designing resource-aware software

The *optimizer* components modularize the optimization strategies; for this matter they interact with both functional and user components.

After identifying and modeling the components, their interface can be modeled. As explained in chapter 6, our notation supports the specification of both *service* and *resource* interfaces for the components.

When the service and resource interfaces are specified, the *resource behavior* of the components can be modeled. The resource behavior represents the dynamic resource consumption and provision of components during their execution, and we represent them via RUMs.

Our workflow prescribes checking activities after each modeling step. Typical checks are (a) checking whether any component is missing, (b) checking whether the required interfaces of components are bound to compatible provided interfaces of other components, and (c) checking the safety and liveness of the models. If any inconsistency is found, the initial activities are performed again in a new iteration to resolve the inconsistency.

When at least an initial version of all models exists, the effectiveness of various optimizer components—in terms of the reduced energy consumption—can be analyzed. When the models are sufficiently precise for this analysis to be performed, designers can select an optimizer component based on the analysis' results. To decide which details must be part of the RUM, there are two concerns:

- The RUM must describe the relation between the services and resources that are on the interface of its component.
- We must know which properties we want to prove. Therefore, these properties need to be identified first.

With respect to these properties, the RUM should be a *minimal* abstraction. That is, it should contain the fewest number of states while still providing enough information to show that the key properties hold, because reducing the number of states generally increases the number of accepted traces when we consider over-abstractions. A minimal RUM has the following advantages:

LESS IMPLEMENTATION DEPENDENT The implementation can be exchanged with any implementation that satisfies the RUM. A minimal RUM specifies fewer constraints, so more implementations satisfy such a RUM.

EASIER TO UNDERSTAND A small RUM is easier to fully understand by human developers.

FASTER MODEL CHECKING Model checking does not scale well to large models. In practice, it is only feasible on small models. Thus, keeping the RUM small is important to allow the models to be checked.

7.2 DESIGN ACTIVITIES IN DETAIL

This section explains the details specific to our approach of the activities of the workflow outlined above. We skip details about the design of functional components and services,

since this is an activity commonly carried out in all component-based software development approaches. We start with the analysis performed throughout the design process and motivate from there the other activities and related models.

7.2.1 Analyze Resource Behavior

Specific to our workflow is the analysis of the system's resource behavior using RUMs. This can be used to analyze which optimization results in the least resource consumption. Based on this analysis, designers can select a composition for the final system design using the most suitable one among alternative optimizer components. The RUMs facilitate *understanding* the resource behavior of components, and analyzing the resource consumption of an application is essential for reducing the energy footprint of the software. This is where RUMs can provide valuable insights during software design or redesign.

A system is composed of multiple components. Since there is a RUM for each component, the overall behavior of a system is described by a composition of these RUMs. Such a composition of RUMs easily becomes complex, which makes the manual analysis of RUMs difficult and error-prone. Therefore, we claim that the analysis must be automated by means of suitable tools. One of the reasons that RUMs must be defined formally, as explained in chapter 6, is to allow tools to reason about RUMs.

7.2.2 Model Resource Ports of Functional Components

Both software and hardware functional components can provide and consume resources, while the kind of resources may be different for hardware and software components. For example, a hardware device consumes electric energy as resource, whereas a software component may require a network connection or a buffer as a resource.

To be able to analyze resource consumption of components, the resources that are provided and required by the components must be specified as the interface of the components. In addition, the inter-connection among the components must be specified so that a provided resource can be consumed by other components. For this matter, our workflow proposes the following approach for modeling the resource ports:

- First, identify a resource with respect to which the software should be optimized and the components that directly require this resource.
- Second, add the corresponding required resource ports to these components. For example, if the energy consumption should be optimized, components that directly use hardware components receive energy as a required resource port. Components that consume the resource of interest in general also provide resources that relate to the resource of interest (e.g., because they consume the initial resource in order to provide another resource at a higher abstraction level). Such related resources must be added as provided resource ports to the components.

This activity must be applied recursively until all relevant resources are identified.

7.2.3 Model Resource Behavior of Functional Components

After modeling the service and resource ports of the components, we need to specify the resource behavior of components, which represents the dynamic relation between the service and resource ports. This dynamic relation can, for example, include the following information: which resources are required or produced by each service port; the amount of the resources; how the amount changes during the execution of a service; how switching among the services influences the resource consumption and provision; etc. A RUM allows representing the resource behavior of components.

RUMs can be created and added to components through the following activities:

- Identify states with distinct functional behavior of a component, for example, by considering the functionality defined as service ports, or by referring to the requirements or hardware specification.
- Define transitions between these states that can happen at events related to service invocation and possibly other events, such as time-outs.
- Add the resource consumption to the model by annotating each state with the amount of resources it consumes or produces; thereby refer to the required and provided resource ports of the component.
 - Split states with multiple characteristics of resource usage into multiple states.
 - Identify the possible transitions between these new states and the already existing states, and events triggering these transitions.

This step must be applied recursively until all states have a single characteristic of resource consumption.

7.2.4 Model User Components

The overall resource consumption of a system and the effectiveness of an optimization strategy can be largely influenced by the way in which the system is used. In general, different kinds of users exist, ranging from human users to other systems; we uniformly use the term *user* to refer to any kind of client actively using the system under design. For example, if a media-player application adopts a caching mechanism to reduce the amount of network connections, caching would not be effective if a user disturbs the normal stream of video by seeking forward and backward. Therefore, to be able to analyze the effectiveness of an optimization strategy, it is needed to model the usage scenarios.

Since it is not possible to foresee all potential usage scenarios, it is desirable to at least model most common scenarios. In our workflow, we represent these scenarios as user components. All functionality that is invoked during the scenario is modeled as required service ports. Based on these services, a RUM is designed to describe how these services are used, as discussed in section 7.2.3.

The simplest specification is a user who might do anything non-deterministically. The RUM of such a user is a non-deterministic state machine that has one state with self-loop transitions for every service that can be invoked. Such a specification can be useful for

analyzing the correctness of the system, but—in general—is not useful for analyzing the resource consumption, because it does not correspond with actual behavior.

In contrast, to facilitate analysis of resource consumption, we propose to model deterministic user behavior. It is possible to specify multiple different user behaviors as alternative components. Then distinct analyses can be performed using each of the available user components to identify the usage scenario under which a given optimization performs best. Non-deterministic user behavior can be modeled in the same way, but increases the complexity of the analysis.

7.2.5 Model Optimizer Components

The optimizer components implement the functionality to monitor and adapt the resource consumption of functional components during execution. For this purpose, they interact with functional components. The optimizer components can be regarded as special kinds of functional components, too. Nevertheless, our workflow distinguishes them from other functional components to emphasize that they must be separated from the other components so that various optimization strategies can be used interchangeably.

In the absence of an optimizer component, all functional components interact directly. There are two main kinds of components: (a) *high-level* components are those directly controlled by the user behavior and (b) *low-level* components are those that directly consume the resource to be optimized. An optimizer component should be inserted as an indirection to the interaction between low-level and high-level components.

There can also be intermediary components through which the high and low-level components interact. When an optimizer component is inserted between such intermediary components, the components closest to the high-level components play the role of the high-level components; this is analogous for the low-level components.

The optimizer component must have as provided interface the services and resources that are required by the higher-level component; the required interface accordingly mirrors the provided services and resources of the lower-level component. The optimization logic must be modeled as the component's RUM following the guidelines in section 7.2.3.

As for the user behavior, it is also possible to design multiple alternative optimizer components. In the analysis phase, various combinations of user behavior and optimizer components can be analyzed. In that way, the designer can identify which of the optimizer components is optimal for which expected user behavior.

7.3 WORKFLOW USING UPPAAL

Even though the workflow presented in section 7.2 can be applied manually, for automatic analyses a tool must be used. We could either develop a dedicated tool to analyze resource consumption or use an existing tool. The advantage of a dedicated tool is that the tool will support exactly our notation, the disadvantage is that the development of a tool is error prone and time consuming. The advantage of using an existing tool is that it can be used immediately, the disadvantage is that we may need to adapt our models to the tool, which

may have negative influence on the modularity of the models. We decided to use an existing model checker, which ideally fulfills the following requirements:

1. It allows expressing service invocation, resource consumption and timing behavior.
2. It allows expressing components that
 - a) can have service and resource interfaces.
 - b) can easily be exchanged by other components with the same interface.
 - c) can be structured hierarchically.
3. Preferably, it can calculate which configuration has the lowest resource consumption.
4. It provides a graphical user interface (GUI) that closely resembles our notation.

UPPAAL [7] is a robust tool to model, verify, and validate timed automata, which can represent power state charts. Scenarios can also be modeled as timed automata. In this section, we present our concrete experience with using UPPAAL; however, other model checking tools can be used as well.

7.3.1 Overview of Uppaal

Here we give a brief and simplified explanation of the concepts in UPPAAL and map them to the terminology used in this thesis. In UPPAAL, a system is modeled as several timed automata that execute in parallel. A *timed automaton* is a finite-state machine with numeric and clock variables.

Transitions can have a so-called *synchronization label*, which has as meaning either (a) the transition can be triggered when the event corresponding to the label occurs, or (b) the corresponding event is emitted when the transition is taken. Transitions can be annotated with *guards*, which are side-effect free expressions; a transition is only enabled when its guard evaluates to true. Transitions can also contain *assignments* that update variables.

States can be annotated with side-effect free expressions, called *invariants*. They restrict the time during which the system can be in a certain state. So-called *committed states* can be used to represent actions that must occur immediately after each other. A committed state cannot delay, and the next transition must involve an outgoing transition of at least one of the committed states in one of the parallel automata.

Figure 26 provides an example of a UPPAAL automaton for a coffee machine that provides coffee for two coins. A coin can be inserted, which increases the balance, specified by the synchronization label `coin?`—the question mark shows that the coin is an input—together with an assignment to the numeric variable `balance`. The start state contains an invariant, `balance ≤ 2`, that ensures that at most two coins can be inserted. After inserting two coins, coffee will be provided unless the refund button is pressed before. (Note that there is no timing constraint, so the coffee machine can postpone providing coffee forever.) When the refund button is pressed, the coffee machine immediately returns all coins, specified by the transition with synchronization label `coin!`—the exclamation point shows that the coin is an output. Since the state that returns coins is committed (denoted by the C), no time can pass in this state, so coin refund is immediate.

Figure 26: Simple UPPAAL timed automaton for a coffee machine

7.3.2 Design Activities

Each design activity in which we use UPPAAL is described below.

Analyze Resource Behavior

After designing a model, the designer can simulate behavior in UPPAAL by stepping through the diagram semi-automatically, which is the simplest form of analysis that UPPAAL facilitates and is mainly useful in the early stages of development. When the state charts become larger, it becomes more useful to let UPPAAL fully do the model checking. UPPAAL provides a subset of timed computation tree logic (TCTL), in which many properties can be expressed, such as safety and liveness properties. For example, we can check whether our model is free of deadlocks. Whenever a given property does not hold for every execution sequence, UPPAAL provides a counterexample that can be used by the designer to analyze whether the model or the property specification is incorrect.

An example of a *safety* property that can be expressed in UPPAAL is: whenever the user presses the play button, the system will eventually start playing music. This property is written as: $A[](\text{User.pressed_play} \implies (A<> \text{MP.playing}))$, or as the shorthand: $\text{User.pressed_play} \dashv\vdash \text{MP.playing}$.

When we have specified a scenario of expected user behavior, UPPAAL can analyze the behavior of the system and output a trace of the expected behavior. Such a trace contains the resource consumption over time. Thus, by analyzing various optimization components, the optimization that consumes least resources can be identified.

Model Resource Ports

The structure of the application is modeled in UPPAAL by creating an automaton for each component. In UPPAAL, variables and labels can be local to an automaton or global to all parallel automata. All services and resources that are on the interface of a component are globally accessible. Therefore, we define global labels for all services and global variables for all resources present in the interface. Provided resource ports become variables that are written to, required resource ports become variables that are read.

Model Resource Behavior

The contents of the automaton defined to model a component, represent the resource behavior. Such automaton can be designed using the steps presented in the general approach; we only explain the UPPAAL-specific details here.

Every transition that changes the consumption of a resource by the component is annotated with an assignment that updates the value of the variable that represents the resource to the current consumption. The resource behavior is linked to the functional interface by adding synchronization labels to transitions. For example, when *play* is defined as a service on the interface, the label *play!* represents requiring (invoking) the service and *play?* represents providing the service.

Model User Components

The behavior of a user can also be modeled as an automaton. A non-deterministic user can be modeled in UPPAAL as elaborated in the general approach (see section 7.2.4). Scenarios where the user performs certain actions after a specified amount of time can be modeled by adding timing constraints to transitions as guards and to states as invariants.

Model Optimizer Components

The definition of an automaton in UPPAAL is actually a template, so that the automaton must be instantiated before it can be used. This enables providing alternatives. When alternative components are desired, multiple components with the same interface can be defined. Then, replacing one component with another is as simple as replacing one statement, namely the instantiation of the component. Since templates can take parameters, parameter-based adaptation is just as simple, namely, to define a component that takes a parameter and pass a value to it during instantiation.

7.4 MEDIA PLAYER EXAMPLE

As an example we describe how we applied our workflow to the media player that runs on a smartphone, which was presented in section 6.4.2. We started with the high-level structure model of the relevant software components of the smartphone, including the media player. Figure 27 shows the part of the model representing the media player component. Gradually, we formalized the resource behavior—shown informally in figure 27—until we ended at the model shown in figure 28. Such a model can be created by starting with the states of the high-level model and iteratively adding more detail, while ensuring that every model is an over-abstraction of the media player. These details are additional states and transitions that formally describe the resource behavior textually present in the high-level model. Such details are added until all relevant resource behavior is added.

Because this formalization was done using UPPAAL, it was possible to analyze the consistency and completeness of our model using TCTL formulas. With the help of this analysis, we gradually added more detail to the model. This was performed in the following five steps,

Figure 27: Media player high-level diagram

which are iterative because adding detail to one part of the model might require adding detail to other parts as well.

SPECIFYING THE USER Verification requires all components to be specified. The high-level structure diagram did contain most components already, except for the user. Thus, we added a model of a simple, non-deterministic user that might do anything.

SPLITTING THE STATE DIAGRAM The high-level state diagram in figure 27 represents the resource behavior of the system, contains various concerns; for example, buffering songs and playing songs. To maintain the modularity of the media player at the detailed level, we need to identify the various concerns and split the state diagram into multiple diagrams, which are hierarchically composed. To split the state diagram in a *player* and *buffer* process, an interface to communicate between these processes must be defined. For instance, we defined the events *start buffering* and *stop buffering*. The player component can generate these events to influence the buffer process.

ADDING DETAIL TO THE STATE DIAGRAM The high-level state chart in figure 27 did not contain enough information to analyze the resource consumption. For example, it did not specify in detail when buffering was performed. Therefore, we added details to the state diagram until we were able to analyze the resource consumption. Most importantly, we had to check that the model was correct and complete. For example, to check that the player actually stops playing when it cannot download music in time and to check that there are no situations for which no possible behavior is defined, i.e., deadlocks.

Figure 28: Media player detailed diagram

ADDING DETAIL TO THE INTERFACE Figure 28 shows that we also augmented the interface of the media player with two services: the required service *no music* and the provided service *downloaded*. These two services do not correspond to the component's functional behavior; instead, they are required to express assumptions about the component's internal behavior that are made by the RUM. Since the RUM is an essential part of the component's interface, it is legitimate and desirable to also promote these details to the component interface in terms of provided and required ports.

The required service *no music* represents the occurrence of an error in the media player component, namely that no music can be played because of the lack of data. This service does not have to be connected to any other component. (Requiring an unconnected service is an error, but since *no music* is an error state anyway, leaving it unconnected does not change the behavior of the media player.) However, explicitly modeling *no music* has the benefit that it provides us the possibility to reason about the situations in which this error condition occurs. Thus, our model of *no music* is a pattern that can be used for modeling error conditions: model errors by explicitly communicating the error through the interface as a required service.

The provided service *downloaded* is added to model that downloading music is asynchronous. Thus, *downloaded* is the callback event that is fired when a download (started using a *download* event) is finished. Since state machines always run in parallel in UPPAAL, the general pattern for an asynchronous request is to create a state machine that handles this request, is started using a *start?* event, and communicates the result back using a *result!* event.

SELECTING THE MOST SUITABLE OPTIMIZER COMPONENT To test various configurations, we created the following three controllers, which play the role of optimizers:

1. Basic controller providing the unoptimized behavior.
2. Burst download controller that downloads as much music as possible at once and then waits until the next burst of music is needed.
3. Fast Dormancy controller that downloads music in bursts and switches the network manager to idle when such a burst download is finished.

Each controller is a mediator between the media player and network manager, as can be seen in figure 29. These controllers were created in the listed order. When we added the second controller, we only had to replace the controller and connect the previously unconnected resource *buffer*, since the controller uses the knowledge of the buffer size to perform its optimizations. The third controller also did not require any changes to other components. This controller uses the *demote* feature of the network, which was already present, but not yet connected. Thus, adding the third controller also required adding only a single connection.

To test the behavior of the system, we created a scenario (partly shown in figure 30). As adaptation we used the three different controllers described before, while running the scenario. The energy consumption during these runs is shown in figure 31. We see that the first controller consumes the most and the third controller consumes the least energy. More details on the experiment are available in appendix D.

Figure 29: Media player overview

Figure 30: User behavior (partial scenario)

Figure 31: Analysis of scenario

7.5 TRADE-OFFS

7.5.1 General Workflow

When carrying out the general workflow, one of the challenges is how much detail should be added to resource utilization models. Since these models are part of the interface of a component, it is desired to hide some of the complexity of these models. Components should hide their implementation details from other components. Therefore, resource utilization models should also hide all implementation details that are not needed to explain the resource consumption. However, in our examples, the resource utilization models express quite some implementation details in order to specify the resource behavior as precisely as possible. This increases the coupling: controllers that depend on the precise resource behavior given in the RUM might not be applicable to implementations that have different resource behavior. Therefore, it might be desirable to add less detail to the RUM, for example by specifying only which power states exist, not precisely when these states are activated. In such a case, the activation of the states should be communicated to the controller at runtime in order to allow optimization. In this way, controller components are more modular, which might be desired even though it could hinder some optimization strategies.

7.5.2 Uppaal-based Workflow

When performing the case study, we had a large benefit from UPPAAL, but we also learned some disadvantages of using UPPAAL. Even though UPPAAL provides a good interface for

entering state charts, it does not match our notation perfectly, as described in the following three paragraphs.

The models used by UPPAAL are not as modular as desired: models cannot be composed hierarchically and interfaces cannot be visualized. In figure 28 we see that one component (the Media Player) consists of three other components. We draw this hierarchy by hand, including the interfaces of all four components. However, it would be possible to automate the conversion from hierarchical models to UPPAAL models: David, Möller, and Yi [30] showed such a conversion using a subset of UML for the hierarchical models.

UPPAAL does not provide a specific abstraction for *resources*. Instead, it provides *variables*, which—among other uses—can be used to represent the resource consumption of relevant resources. Therefore, figure 28 does not contain any resource ports, but it contains black diamonds to denote constants and white diamonds to denote variables.

UPPAAL can analyze the quality of a certain composition, but cannot choose the best composition, because each analysis is independent of the others. Analyzing alternative compositions and deciding which one is best, must be done separately, either by hand or using another tool.

This chapter describes how RUMs can be extracted automatically from source code, based on our publications “Deriving Minimal Models for Resource Utilization” [P.10], presented during the 1st International Workshop Green In Software Engineering, Green By Software Engineering, and “A Tool-Supported Approach for Modular Design of Energy-Aware Software” [P.11], presented at the 29th Annual ACM Symposium on Applied Computing.

Modeling the RUM of components is a challenging task and the expressiveness of the RUM language determines the quality that can be achieved in the subsequent analysis. During software design, the implementation of third party components is often already available. Therefore, this chapter focuses on modeling RUMs for the *existing* identified components, i.e., components that can be re-used in the design project at hand. For example, when developing an application for playback of media from the Internet, the Network Manager component may already exist. Thus, of the activities comprising our design method presented in figure 32, which are the same activities that were shown in figure 25 on page 78, this chapter mainly focuses on those with a grey background. We focus only on two kinds of components that are identified in our methodology:

- The *functional* components refer to both software and hardware components that form the target system.
- The *optimizer* components modularize the optimization strategies; for this purpose they interact with functional components. In general, components may be nested and as such an optimizer component may also be contained in a functional component.

After identifying and modeling the components, their interface must be modeled and their *resource behavior* must be defined as a Resource-Utilization Model (RUM). This chapter presents an approach for extracting a RUM from implemented components, based on the formal method of *counterexample-guided abstraction refinement* (CEGAR) [24].

8.1 TOOL SUPPORT

A software system can be specified as a composition of components, which easily becomes complex, and beyond the capabilities of existing model checkers to handle directly. Thus, it is necessary to create an appropriate over-abstraction that is much smaller than the full model, and only includes the information directly needed to establish a property, for each desired property to be asserted. Manual analysis of such a composition and finding the needed over-abstraction is difficult and error-prone. Therefore, we claim that both the analysis and creating the over-abstraction must be automated by means of tools.

Figure 32: Design method for resource-aware software

Figure 33: Steps of CEGAR

This chapter explains how creating RUMs can be automated by using the MAGIC tool [18, 19]. However, our method does not prescribe that this particular tool must be used. To allow the use of other tools that support the same principle, we also explain the underlying CEGAR method on which the tool is built.

8.2 CEGAR

Counterexample-guided abstraction refinement (CEGAR) is a formal method to semi automatically refine abstract models based on counterexamples, when a concrete model is available. The purpose of CEGAR is to increase the scalability of model checking by creating and using abstract models that contain the minimum amount of detail needed for the desired model checks, instead of using the generally much larger concrete models for these checks. It has been implemented in several tools [11, 25], most relevantly in MAGIC for verifying systems composed from multiple parallel components [18, 19]. It has also been applied to events and aspects [32]. In short, the application of CEGAR requires:

- Specified properties that are being checked on abstract models; a violation of such a property leads to a counterexample.
- A detailed concrete model, from which information can be added to the abstract model.

The abstract models used by CEGAR must be over-abstractions of the concrete model. When CEGAR refines an abstract model with details from the concrete model, it ensures that the refined model is also an over-abstraction of the concrete model. Thus, CEGAR generates models that are suitable as RUMs.

Abstract models are refined through the steps shown in figure 33. First, an initial abstract model is derived, usually by simple static analysis of the concrete model, but guaranteeing that an over-abstraction is used. When CEGAR cannot prove a given property on the abstract model, an abstract counterexample is produced. This failure to prove the desired property could be due to two scenarios:

- There is *no* real error, but the abstract model does not include enough information about the concrete behavior. In this case, the abstract counterexample produced for

the desired property does not correspond to an actual error in the concrete system (and it is called a *spurious* counterexample) and CEGAR can be used to automatically refine the abstract model.

- There *is* a real error, and then an inventive step is needed. An option is to weaken the specified desired property. But it may also be the case that an error in the implementation is found, which must be fixed by the responsible developer. This step can be guided by the counterexample, but is not automatic.

Using CEGAR requires (a) simulating the steps of the abstract counterexample to check whether they correspond to any execution of the concrete system. If not, (b) (minimal) information can be automatically extracted from the concrete model to produce a refined abstract model in which the previous counterexample cannot occur, and then the tool should again attempt to verify the desired property on the refined abstract model.

The key steps in CEGAR use algorithms from model checking. The simulation requires showing that no concrete execution that corresponds to the abstract counterexample is possible. At the same time the key information that makes the counterexample invalid in the concrete model must be extracted. This involves finding the core conjuncts that make a complex Boolean expression unsatisfiable. This is today usually done by exploiting the progress made in solvers that determine whether a complex propositional formula with thousands of conjuncts is satisfiable (a SAT solver) or solving extensions to richer formulas known as SMT solvers. The formula constructed is an encoding of an assertion that the steps in the concrete model and steps in the abstract counterexample are possible together. When this is true, the abstract counterexample represents an actual error in the concrete system. When the check fails, the SAT (or SMT) solver provides a so-called *SAT-core expression* that has the fewest conjuncts showing why the formulas cannot be true together (see Clarke et al. [25] for more details). From this, the minimal information that should be added to the over-approximation can be automatically derived, without any user intervention. This gives a refined model for which the previous abstract counterexample cannot occur, and then a new attempt at model checking can be done on that model.

8.2.1 Example of Using CEGAR

As an example, we consider the design of the media player presented in section 6.4.2 for playing music from the Internet. Significant energy will be consumed by the network component within the phone. The overall system's energy consumption can be optimized by steering the media player to stream the data in bursts operating at full bandwidth and store it in a buffer; when the buffer is filled, the network device can go to powersave mode until the buffer runs empty and another data transfer is necessary. For this optimization to be effective, it must be possible to predict at which rate the buffer is emptied depending on the playback rate, and it must be known how quickly the network device can switch between powersave and active modes, in addition to the amount of power consumed in both modes.

The example application consists of three components: a Media Player, Optimizer, and Network Manager. This application is a concrete example, but our description is applicable to any system that consists of an application component, an optimizer and a low-level—or

Figure 34: Simple over-abstraction

hardware—component. It can also be generalized to applications that consist of more than three components.

To design this media player, we must design a RUM—an abstraction—for each key property of each component. Once we have these RUMs, we can analyze whether this abstraction entails the desired behavior. If it does, we have defined a generic optimizer: it guarantees the key property for any concrete media player and concrete network manager as long as both satisfy the abstract version on which we performed the checks. However, we are still left with the question how to design these abstractions, which will be elaborated in the remainder of this section.

As noted above, our application has Media Player, Optimizer, and Network Manager components. We also assume that we already have a concrete model of the Network Manager, because this is a fixed low-level component that is already available. Presumably, the designers of the network manager analyzed its resource behavior and created a detailed model of the network manager, which contains at least all information that is relevant to our media player. This concrete model might be the actual implementation of the network manager together with the relevant energy consumption of this implementation, or a model that contains too many details to allow practical model checking of key properties. Since we did not design this model for the purpose of designing a media player or guaranteeing its properties, it probably contains much more information than needed for this purpose. Therefore, it is not a minimal abstraction, and is not directly useful for the design of our application.

The identification of the minimal abstraction and the degree to which detailed energy information must be included in it is a challenge. Let us start with a very abstract model of the network manager, e.g., just a single state that restricts the power consumption below a value that is at least as high as the maximum power of the concrete implementation (an example is shown in figure 34), and refine that to a RUM with just the information needed to justify the key properties of the system. This single state must be an over-abstraction of the component itself; that is, even though the model contains too little information to be useful, it must represent all correct behavior of the application.

For example, stating that the power consumption in this single state of figure 34 is less than 1 J/s is fine, since the network manager *never* consumes more than 1 J/s. However, stating that the power consumption is larger than 0.5 J/s, for example, is simply incorrect, because in fact the network manager consumes less than that when it is idle.

When we have such an abstraction, we can use key properties to automatically refine this abstraction with the help of CEGAR to something more useful. Key properties are properties that should hold for *every* execution sequence. That is, any linear temporal logic (LTL) [65] formula can be used to specify a key property. In general, key properties are a combination of both safety and liveness properties. Examples of key properties are (a) uninterrupted operation (safety), (b) the amount of energy consumed is within certain bounds (safety), and

Figure 35: Slightly more detailed abstraction

(c) pressing the play button should eventually result in playing music (liveness). Note that without liveness properties, totally shutting down the system might be the best optimization: no energy is consumed and nothing can be interrupted. However, a media player that never plays any music is not useful at all; therefore, key properties are generally a mix of both safety and liveness properties. For the media player, we specify the key property: “in every execution sequence, the media player consumes less than 10 J for playing 20 s of music.”

With the current abstraction (shown in figure 34), we can only show that for 20 s of music, less than 20 J is required. Thus, consuming 18 J during 20 s of playing music is a counterexample for the given key property, because it does not satisfy the key property and is a possible execution sequence according to the current abstraction. Therefore, to show that the property holds, the abstraction must be refined.

Assuming that the given property is indeed satisfied in the concrete model (because the network manager switches to idle when it is not downloading music), CEGAR can identify that this counterexample is spurious—it does not hold in the concrete model. CEGAR simulates all concrete events that correspond to a self-loop in the abstract state with a power consumption of 0.9 J/s for 20 s and discovers that no concrete execution corresponds to this abstract execution. Based on the concrete model, CEGAR can automatically create a refinement of the abstraction shown in figure 34.

CEGAR, and in particular its implementation in the MAGIC tool, can identify the key information needed to show that the abstract counterexample is spurious for the concrete system (so no concrete execution sequence corresponds to it). Then, this information is used to build the refined abstraction shown in figure 35, by adding more concrete information on the actual power needed during download, and splitting the abstract state into a downloading state, and an idle state with low power consumption when the request has been answered. The predicate *done* becomes true when the network senses that the request has been answered. This abstraction is detailed enough to show that for playing 20 s of music, less than 10 J is needed: so a new attempt to model check this property will succeed.

8.3 USING MAGIC FOR DERIVING RUMS

We experimented with several tools that implement CEGAR. Because of technical difficulties with some of the tools, we have extracted all models with MAGIC.

Figure 36: State of refined model generated by MAGIC

MAGIC [18, 19] is a tool for automatic verification of C programs against finite state machine (FSM) specifications. MAGIC follows the CEGAR method and uses C source code as concrete model. First, MAGIC extracts an initial abstract model from C source code using predicate abstraction and theorem proving [20]. Subsequently, this model is refined until either it contains enough detail to show that the specification holds, or a real counterexample is discovered. MAGIC either outputs a success message or the concrete counterexample.

MAGIC can output all models it creates, which are over-abstractions of the source code. The last model MAGIC creates in the success case contains sufficient information to prove the key property. Thus, we use this model as RUM. The size of these models created by MAGIC does not directly depend on the size of the source code, it only depends on the size of the code that is concerned with relevant properties. In our example, when reasoning about energy consumption is related to inactivity time, the size of the model depends on the number of locations where the inactivity timer is updated, not on how much data is written to the connection in between.

Figure 36 shows a state of a model generated by MAGIC. Such a state contains a set of predicates, which are all true while the system is in that state. MAGIC uses C code as notation, so the Boolean value *true* is represented by the number 1 and the Boolean value *false* by the number 0. The arrows represent the possible transitions between the states.

8.3.1 Automatically Simplifying Derived RUMs

Even though the models created by MAGIC only contain details that are related to the key properties, the models are larger than the size we envision for RUMs. This is mainly because MAGIC only wants to keep the models small enough to perform model checking, but we would ideally keep them small enough to be human readable, which is much smaller. Therefore, we post-process the models generated by MAGIC to reduce their size, using the transformations outlined in the following paragraphs (this section describes the concepts, technical details are provided in appendix E).

First, we decide which variables we consider important. For example, when we want to know the relation between inactivity time t_1 and energy, the variables *inactivity*, t_1 , and *energy* are important. For every state, we remove all predicates that do not contain a variable

Figure 37: Excerpt of simplified over-abstraction

of interest, to hide unimportant details of the system. This does not reduce the number of states, but—in general—creates many states that are similar.

Second, we simplify the predicates of each state. Using the CVC3 or CVC4 theorem prover [5, 6] we identify both (a) predicates that are always true because they are implied by other predicates, and (b) parts of disjunctions that are always false because they contradict other predicates, and remove these from the predicates. For example, the state shown in figure 36 is equivalent to the rightmost state of figure 37.

Third, we reduce the number of states by merging equivalent states. We do this by calculating the stutter bisimulation quotient [4]. Stutter equivalence considers state changes invisible if the truth value of the predicates does not change, that is, if two states have exactly the same predicates. Thus, the stutter bisimulation quotient is a smaller transition system, because it removes such invisible transitions.

8.3.2 Application of CEGAR

The media player of our example consists of several components, among which is a network manager. This network manager is not designed specifically for the media player, but is a component that existed before the media player was designed. Therefore, it is desirable to automatically create a RUM for the network manager, based on its source code.

To show that this is possible, we have written a simple network manager in C. The key property we used is that as long as the inactivity timer is less than t_1 , the network manager consumes no more than 300 mA. This key property is a *safety* property that must hold for *every* execution sequence, since it states that for *every* execution of the system, the energy consumption is *always* below 300 mA while the inactivity time is below t_1 . In MAGIC, this key property K can be defined as follows:

```
K = (
  {inactivity = [$0 < t1 && energy <= 300]} -> K |
  {inactivity = [$0 >= t1]} -> K
).
```

Property K states that whenever *inactivity* changes to a value strictly less than t_1 , the energy consumption must be less than 300 mA and afterwards K must hold again. If *inactivity* changes to a value greater than t_1 , there is no requirement except that K must still hold.

MAGIC can indeed prove that this key property holds and provides us with an over-approximation of 429 states. We simplify this over-approximation with a script that performs the simplifications described in section 8.3.1. This simplification outputs an over-approximation of 60 states. Thus, a single key property results in a model that is reasonably small; nearly small enough to be human readable.

8.4 MODEL CHECKING RUMS

After creating the RUMs of all components, the resource utilization of the whole software system is specified by the composition of these RUMs. This composition consists of many parallel state charts, which is too large to easily analyze by hand. Therefore, using a model checker is necessary.

It would be nice if creating RUMs and model checking them could be performed with the same tool, but unfortunately we have not found a tool that facilitates both. MAGIC focuses on automatically extracting models, but does not provide the ability to manually create models for newly designed components, such as the Media Player component. In section 7.4, we showed how UPPAAL can be used to provide additional analysis of RUMs. This shows that UPPAAL indeed facilitates analyzing energy consumption based on RUMs. However, the RUMs used for that analysis were created manually; the RUMs extracted by MAGIC cannot be automatically converted to UPPAAL models.

8.5 RELATED WORK

A wide range of techniques and mechanisms for making software energy-aware are being proposed. These are usually dedicated solutions or frameworks for facilitating optimizations, for example, at the level of operating systems [108], at the level of compilers [27], or at the system level [98]. Recently, more emphasis is put on modeling and analyzing energy-consumption at the application level.

The need for reflecting energy consumption of software in design models is also studied by Sahin et al. [87]. However, in contrast to our approach, no concrete method or tools are provided to model resource utilization of components, to (automatically) derive RUMs and to analyze them.

Götz et al. [47] propose a model-driven component-based approach for software systems that can be optimized w.r.t. their provided quality and energy consumption at runtime. In this approach, operational modes of components and the resource utilization in each mode are specified as a state machine at the interface level. The optimization problem is defined as an integer linear program (ILP); an ILP solver determines an optimized software configuration. We identify two difficulties in adopting this approach. First, manual identification of components' resource behavior can be error-prone, and second, this approach assumes that

the implementation of the components fulfills their specified resource utilization, where this may not be the case in reality. We overcome these by automating the derivation of RUMs from concrete models (e.g., their implementations) by means of key properties. Since key properties are checked in the models during the derivation process, it is possible to identify that the concrete models fulfill the desired resource utilization.

Deriving detailed information about the energy modes of applications and the amount of energy consumption in each mode is a challenging task. In our approach, we assumed that information about energy/resource consumption is available in the implementation as annotations over which we can evaluate key properties. Several approaches, discussed below, investigate tools to obtain energy models of software. The accuracy of these models depends on the granularity of instrumentations to obtain energy consumption information. Hähnel et al. [48] utilizes Running Average Power Limit (RAPL) energy sensors available in recent Intel CPUs to measure the energy consumption of short code paths (e.g., individual functions). Powerscope [42] facilitates profiling the total energy consumed during a certain period in each process and/or procedure. Eprof [75, 76] is a fine-grained energy profiler for smartphone apps, which instruments the app source code for tracing the energy consumption of system calls and application calls. As a result of tracing, Eprof generates a finite state machine depicting various energy modes and the energy consumption of apps.

The above-mentioned approaches profile the energy consumption for a subset of the behavior of applications, i.e., the behavior that is executed. Hence, the derived energy models are under-abstractions, and as we discussed in section 7.2.3, by adopting under abstractions we cannot provide guarantees over the overall resource utilization of the applications. Therefore, these approaches can be regarded as complementary to ours; we can use the energy models resulting from these approaches to annotate RUMs with more accurate information about the energy consumption of each provided/required service of components.

The *eLens* approach [49] combines per-instruction modeling with program analysis to create fine-grained estimates of energy consumption. However, their approach does not consider abstractions or the existence of various power states. Therefore, our approach is more suitable for large software systems, but the fine-grained estimates of *eLens* might be usable as energy input for our approach.

Extending software with energy optimization functionality is a typical way to make software energy adaptive [38, 41, 47]. In contrast to our approach, these approaches fix the optimization functionality, and do not provide means to analyze the impact of various optimizers on the overall energy consumption of software to guide designers to choose suitable optimizers accordingly.

8.6 CONCLUSION

In this chapter, we have shown that by using the CEGAR method, compact Resource Utilization Models (RUMs) can automatically be extracted from existing functional components. These compact RUMs help establish key properties relating to energy consumption. However, manual effort is still required when key properties cannot be proven, either to resolve an error in any of the components, or to add missing detailed information to the concrete

model. Creating new optimizations also requires an inventive step from domain experts. Thus, our approach does not invent new optimizations, but it can help domain experts acquire the knowledge needed for designing optimizations.

For the media-player example, once we have refined the abstraction, we observe that the power states of the network manager coincide with its functionality. Based on that, a domain expert can formulate a possible optimization, for example, based on the knowledge that downloading in bursts consumes less energy than downloading in trickles. Such a property can already be checked on the model of the network manager (as shown in section 7.4) and when it holds, it is an opportunity for optimizing the application.

It could also happen that the power states do not coincide with the functionality, they may even seem arbitrary. In such a case, a domain expert might not find any opportunities for optimization.

In this chapter, we have shown how (formal) tool support for defining RUMs can be provided. We have presented how to use the tool MAGIC and post-process its output to automatically extract RUMs from existing component implementations.

Our approach generates over-abstractions by formally analyzing source code, which is an advantage over testing-based approaches. This is because such approaches can only consider a subset of the possible behavior of software. Thus they cannot give guarantees.

This chapter describes how profiling can be used to acquire energy information while automatically extracting RUMs, as published in the paper “Interpreting Energy Profiles with CEGAR” [P.12], which received a gold medal in the student research competition of the 29th Annual ACM Symposium on Applied Computing.

The previous chapter showed how the CEGAR method can be used to automatically extract models from source code. CEGAR assumes that source code contains enough information to create a model of its energy consumption. Nevertheless, in general, source code does not contain energy information, so an additional source of information is needed.

Another way to acquire energy information is profiling the software at hand. That is, executing the software and measuring the actual energy consumption. By combining profiling with CEGAR, it would be possible to extract RUMs automatically from source code that lacks energy information, as will be explained in this chapter.

CEGAR consists of the four steps illustrated in figure 38. Compared to the previous chapter, we add profiling to the third step of CEGAR. The other steps of the CEGAR loop are still the same: we start with source code, from which we (1) derive an **abstract** model, (2) **model check** the key property, (3) verify the correctness of counterexamples through **profiling**, and (4) **refine** the model with the information acquired during profiling.

Profiling does not provide energy information for every possible execution sequence of the application; each run profiles only a single execution sequence. However, CEGAR expects an *over*-abstraction as the concrete model, which contains at least every execution sequence. Also, energy consumption might be dependent on state outside the application that is being profiled, which is invisible to the profiler. For example, the state of the network driver can influence the energy needed for transferring data over the network. Thus, the energy consumption of a specific execution sequence cannot be acquired exactly with a single run only: we must take into account that profiles are inexact, whereas CEGAR expects an exact specification (an over-abstraction) as concrete model.

Figure 38: Steps of CEGAR

Figure 39: Example behavior

Figure 40: Actual energy consumption

Therefore, our challenge is (a) to combine two models: the source code plus the results of profiling, and (b) to adapt CEGAR to work with uncertain (or probabilistic) data. This chapter discusses how we treat these challenges in the four steps of CEGAR.

9.1 OVERVIEW OF CEGAR WITH PROFILING

We will now give an example that demonstrates typical steps in CEGAR when combined with profiling. Consider that the state chart in figure 39 is the behavior (i.e.: RUM) of an application on which we run our approach. Deriving this RUM started with specifying a key property, which could be: *reaching Z always consumes less than 16 J*. This key property cannot be guaranteed. Therefore, our approach will give a counterexample, which could be the following sequence that consumes 17.5 J (denoted by which states are visited, how long the application stays in these states, and how much energy is consumed in each state): $W, 5\text{ s}, 4.5\text{ J} \rightarrow X, 10\text{ s}, 8\text{ J} \rightarrow Y, 5\text{ s}, 5\text{ J} \rightarrow Z$.

We must validate that this counterexample is spurious and derive all necessary information missing in the RUM from the application. From the state chart, we see which services must be invoked to visit the states of the counterexample. We assume that we can invoke the services *a*, *b*, and *d* at the moments given by the counterexample. This allows us to generate a test case that stays in each state as long as given by the counterexample by executing these services at the right times. Executing this test shows us whether both the timing and the energy consumption of the counterexample are realistic. Figure 40 depicts a possible output profile for this example: it shows the measured energy consumption over time. From this figure we can conclude that in reality, the counterexample did not occur, because both states *W* and *X* consume less energy than the maximum allowed by the RUM in figure 39. Thus, we can be confident that the counterexample is spurious and we refine the RUM with the actual energy consumption. On the refined RUM, the above steps are applied again iteratively, which shows that the key property holds.

Figure 41: Example energy profiles

Figure 42: Precision of energy profiles

In reality, we expect the energy consumption to be more dynamic than shown in figure 40, for example as shown by the graphs in figure 41. Extracting a precise profile may require correlating changes in energy consumption to execution events, such as service invocation. The dashed *max* lines in figure 41 show three possible energy profiles which could be extracted from the same measurement.

The more precisely we can extract the energy profile, the more useful the result will be. We define precision as the difference between the energy profile p and actual energy consumption a : $\int |p(t) - a(t)| dt$, where lower is more precise. (Note that when we profile the maximum, $p(t) \geq a(t)$ must hold, so we can omit taking the absolute value.) In figure 42 the precision is shown as the gray area between the actual consumption (the solid black line) and the profile for the maximum energy consumption (the dashed gray line).

Figure 41a shows the simplest profile: the energy consumption is always below a constant value. However, this value is often much higher than the actual energy consumption, so it is not a very precise profile.

In figure 41b the maximum is reduced at some moment. This creates a more precise profile, but requires knowledge about when the maximum energy consumption changes (e.g.: after a certain time or after invocation of a specific service).

Instead of providing a maximum for the consumption at any moment, figure 41c provides a maximum for the average energy consumption. Since, in general, we are interested in the total energy consumption, having guarantees on the average consumption is sufficient. Even though an energy profile of the average energy consumption provides less information than a profile of the energy consumption at any moment, such a profile can be useful when it provides higher precision. Figure 42c shows the actual average as dotted line and the precision as the gray area between this line and the profile. We see that energy profile 3 provides highest precision.

9.2 DETAILS OF CEGAR WITH PROFILING

This section explains how the four steps of CEGAR (shown in figure 38 on page 105) are combined with profiling. In chapter 8, we used existing tools to perform CEGAR. However, to add profiling to the CEGAR method, we need to tailor this method. Therefore, we created our own tool chain, which will be explained in this section (and in more detail in appendix F).

9.2.1 Abstracting

The CEGAR method starts with extracting an initial abstraction from code. In our case, we use Java bytecode as source code, and extract a model from it with the Java Bytecode ++ (jbcpp) tool¹. The initial jbcpp model is quite large, since it contains *all* bytecodes that are present in the program. Because we are interested in an abstraction, we group sequences of bytecodes, as long as these do not contain any of the following:

1. Method calls
2. Branching

We would like to extract the portion of the model that consumes significant power, because that is what we are going to profile. A single bytecode operation is too short to contain significant power, only a long sequence of such operations should be considered. Method calls are represented by a single bytecode instruction, but do invoke behavior specified elsewhere, which may consume significant power. Therefore, method calls are kept.

Branching influences the path that will be taken, which might be a choice between high and low power consumption. Also, branching may represent loops and repeated execution, of course, may result in significant power consumption. Therefore, CEGAR is interested in conditionals which are represented as branches in the code, such that it can decide which of the paths will be executed. Thus, branching is also kept during abstraction.

Initially, little is known about the energy consumption of the application. The initial abstraction will therefore assume that the application might consume its maximum power consumption at any moment in time. This leads to an over-approximation of the power consumption, which will be refined by CEGAR later.

9.2.2 Model Checking

This step verifies whether the given key property holds on the abstract model. Chapter 7 explains how model checking is supported by UPPAAL. In our tool chain, we also model check with UPPAAL, by transforming the jbcpp models to UPPAAL models. These UPPAAL models are then used to check key properties, for which UPPAAL can generate traces representing abstract counterexamples. Such counterexamples are traces that are interesting to profile.

As explained in section 8.2, the abstract counterexamples are either (a) spurious counterexamples because not enough information is known about the concrete behavior, or (b) real errors that should be fixed by the responsible developer or architect. The goal of

¹ See the SourceForge project *jbcpp* at jbcpp.sf.net.

profiling is identifying which of these is the case and acquiring more information when the former is the case. Thus, the counterexamples provided by UPPAAL are the traces that we profile during the next step of the CEGAR method.

9.2.3 Profiling

For profiling, we use the traces acquired as counterexamples during model checking. However, we cannot profile every instruction present in a trace, because the overhead of profiling is only negligible when the profiled part consumes a considerable amount of time and resources. Since we are trying to create an abstraction that provides bounds on the execution times and resource consumption, we are also only interested in parts consuming considerable time or resources.

The goal of profiling is to acquire an energy profile that is detailed enough to prove the given key properties. Since we cannot profile every instruction of a counterexample, we have to select ranges that are most likely useful to profile. Profiling can be automated, so it is not needed that every acquired profile provides useful information, but the CEGAR loop must be able to reach useful profiles automatically.

We choose loops as starting points for inserting profiling code, as in general it is hard to predict how long a loop will run and how much energy it will consume. Therefore, an over-approximation might initially assume that both runtime and thus also energy consumption are infinite. However, in practice the runtime is bounded, which also bounds the energy consumption.

Thus, we start with profiling the outermost loop present in the counterexample to see if we can acquire energy bounds that satisfy the given key property. Note that we are not interested in the precise number of loop iterations, we are only interested in the maximum (and possibly minimum) power consumed by the loop. If this coarse-grained profile does not provide the desired level of detail, we further analyze inner loops during the next iteration. This way, we only perform the complex analysis of nested loop structures if the loop consumes significant energy. With *desired level of detail* we mean enough detail to show whether the key property is satisfied, as will be explained in section 9.2.4. The profiling step will not evaluate whether the acquired profile does have the desired level of detail. However, each iteration acquires a more detailed profile until the CEGAR loop terminates successfully or with a real counterexample.

After detecting that a given counterexample is spurious, it is easy to refine the model to one that excludes this counterexample. During refinement, however, the goal is to rule out a whole family of counterexamples, not just a single counterexample. Therefore, we have to generalize the model, which is where uncertainty comes in. For example, if the energy consumption of a loop is heavily dependent on the input data, it might only consume little energy for the specific input data used during profiling, but more energy for other input data. Thus, profiling must be performed with various input data until enough certainty is acquired on the resource consumption bounds.

Based on the counterexample, it is possible to automatically detect which ranges of instructions should be profiled. Once this is known, logging can be inserted for measuring entering and exiting these ranges. Since the model contains all bytecode instructions of the

actual application, logging can be added by inserting the bytecode instructions for writing to the log in the model, after which the model itself can be executed. The test driver still might need to steer the execution of the application by issuing input events. Which input events must be generated can also be derived from the counterexample.

When the application is being executed, the resource consumption must be measured with external tools. This can be done with software profilers, such as Trepn, or with hardware, for example provided by the SEFLab (the tools will be explained in section 9.3). These tools can measure the logged events, which is when a certain range is entered or exited, together with the energy consumption over time.

9.2.4 Refining Abstraction

When a sequence has been profiled, the measured time and energy consumption can be generalized to energy bounds, which can be annotated on the sequence. This way, the model is refined with more precise timing and energy bounds, which can be used to validate whether the key property is satisfied or not. If the key property is indeed satisfied (as will be decided by step 2 of the CEGAR loop), the process finishes and the resulting model is the desired one.

If the key property is not satisfied, there are two possibilities: (a) the newly obtained profile violates the key property, or (b) the profiles do not provide enough information to show whether the key property holds. In the former case, a concrete counterexample was provided (by step 3 of the CEGAR loop), which the developer can use to either correct the code or correct the key property, since one of these must be incorrect. In the latter case, we consider the counterexample spurious and the CEGAR loop continues with another iteration of profiling, during which more information will be acquired.

Because of the uncertainty introduced by profiling, the definition of *spurious* counterexamples is slightly different from the definition we used before. In section 8.2, we consider a counterexample spurious if it *cannot* occur in reality. Now, we consider it spurious if the counterexample *did not* occur in reality. Thus, there still is a possibility that a counterexample considered spurious will occur eventually, i.e., is *not* spurious. Profiling cannot prove the absence of counterexamples, but through profiling we can gain confidence that no counterexample can occur in reality. Therefore, the refinement process progresses in two directions:

- When we have enough confidence that the counterexample will never occur, we refine the abstract model such that the counterexample cannot occur anymore (as we already did in section 8.2).
- When we only have little confidence that the counterexample will never occur, we refine the profile with additional runs such that the confidence increases or the counterexample is shown to be real.

The last direction is newly introduced because we add profiling. For now, we decide whether we have enough confidence manually. Automatically deriving the confidence from a set of profiles is still future work.

9.3 TOOLS FOR PROFILING

During profiling, the application is analyzed with external tools while it is running. We have evaluated several such tools and present the tools we consider most promising in this section. Although profiling provides higher accuracy, estimating energy consumption through static analysis can be an alternative to acquire energy profiles faster [1, 63].

9.3.1 Trepn

Trepn Profiler² is an energy profiler developed by Qualcomm. Trepn runs on most Android devices, but the accuracy of power measurements might differ depending on the device, and some features are only available on devices using a Qualcomm Snapdragon processor. Trepn can display data in real-time or store it in a log file for offline analysis [T.4]. It also offers detailed graphs that can help in analyzing how applications use energy. Trepn can analyze one particular application, or the device as a whole. Qualcomm also offers an Eclipse plug-in for Trepn, that developers can use to correlate battery spikes with system events. Using this plug-in, developers can import the data Trepn generates directly into Eclipse, which can make development more efficient.

9.3.2 JouleUnit

JouleUnit [104] is a framework for profiling the energy consumption of Android applications for generating energy labels. To provide such labels, the framework supports writing system tests that execute a scenario and measure the power consumption of the full device. Such measurements are performed by special hardware—e.g.: an oscilloscope—after which the framework synchronizes the measurements with events generated through JouleUnit API calls. Special hardware is required because energy measurements made by the smartphone itself are too coarse grained: generally, software generates events at a higher frequency than the internal battery sensor measures. The special hardware provides measurements that are fine grained enough to synchronize the measured samples with such events.

9.3.3 SEFLab

The Software Energy Footprint Lab (SEFLab) [39] executes software products on a server (or desktop system) and outputs power measurements taken during the software execution. This server is rigged with sensor boards attached to the power distribution lines that go from the power supply unit to the motherboard. This way, accurate independent power measurements are obtained for several components. The measurements are processed on a separate PC, which allows for real-time visualization of the data as well as to store it in file.

The main distinction between the SEFLab and software profilers such as Trepn is that the SEFLab has extensively validated the accuracy of their measurements, whereas software profilers sometimes provide very low accuracy (the actual accuracy is often unknown). Soft-

² See the Qualcomm Developer Network website at developer.qualcomm.com.

ware profilers can provide valuable information for comparative analysis over a period of multiple seconds, but are insufficient for quantitative analysis or short events [53]. Also, of course, the platform that is used for Trepn—Android—differs from that of the SEFLab—server and desktop systems. The kind of data that is acquired from both approaches is similar, so the choice for a specific profiler is not inherent to our method and, therefore, does not limit the generality of our results. For its simplicity and because we perform comparative analysis only, we chose Trepn to experiment with. However, the high accuracy guaranteed by the SEFLab would make the SEFLab our preferred choice for future research.

9.4 EXAMPLE

We have written a Java class that is 72 lines of code and periodically downloads a file from the Internet. When we extract a jbcpp model from this program, we get a rather large model, containing two classes and four methods. This model is shown in figure 43. The intention of this figure is to illustrate the size and complexity of the of the model; therefore, the details are not shown. However, by applying our two transformation steps to (a) group sequences of nodes, and (b) remove unconnected nodes we acquire a much smaller model, as shown in figure 44. This model still contains two classes and four methods, but the number of nodes inside the methods is reduced.

When we translate this model to UPPAAL, we get the model shown in figure 45 on which we can perform our model checks. The UPPAAL model consists of three templates: one for each class and an additional (very small) one that drives the execution.

Model checking provides us with a trace, from which we can acquire the Trepn profile shown in figure 46. The left quarter of the graph differs from the rest, because it includes startup time and, therefore, should be ignored. The rest of the graph shows a repeating pattern, since our trace contains a loop. When we look at the bottom graph, we see the application states; the application switches between two states. In the graph above that, we see that one of these states corresponds to sending data through the WiFi network. This is also the state that consumes most energy, as shown by the graph at the top.

This profile provides enough information to update our jbcpp model with timing and energy constraints, such that the key property is satisfied. So, we see that we can use our toolchain for augmenting extracted models with resource information. Applying it on larger software application remains future work.

Figure 43: Jbcpp model of source code (to illustrate the size)

Figure 44: Jbcpp model after grouping

Figure 45: UPPAAL model

Figure 46: Trepn profile (the graphs represent: battery power, cpu frequency, cpu load, WiFi traffic, application states)

Part III

BACK MATTER

10 | CONCLUSION

Energy is one of many resources that is controlled by software, and reducing energy consumption cannot be considered without taking into account the trade-offs with other resources (e.g., memory and bandwidth usage) and services (e.g., delivered quality of audiovisual artefacts). Our focus is on the reduction of energy consumption by controlling various external hardware components that are energy intensive. Such hardware components are typically represented by dedicated software components, such as device drivers, and are in this way controlled by software.

Software complexity is already problematic and extending software functionality with energy optimizers increases its complexity further. Embedding optimizations into functional components also causes tangling of optimization and functional behaviors, which increases the complexity and decreases the understandability of the software.

Modularization reduces complexity, because it allows reducing the scope of focus to individual modules. A module is a reusable software unit with an implementation and well-defined interfaces, through which modules communicate with one another. Therefore, the interfaces must convey sufficient information to facilitate communication.

10.1 CO-OP

To modularize software effectively, strong composition operators are needed for combining modules. Therefore, the first part of this thesis introduces a programming language, named Co-op, that aims to improve language technology. Co-op can be positioned between AOP and a MOP: it is less expressive than a full MOP, but more expressive than AOP, because as well too expressive as too limiting languages reduce the readability of the source code.

Co-op provides *first-class composition operators*, which allows the composition of both behavior and data. We present a library that provides a diversity of composition operators to show that Co-op indeed facilitates developing first-class composition operators as a library. Most notably, it provides various kinds of data composition that, until now, were only provided as built-in constructs of programming languages.

We outline a solution in which advanced control-flow mechanisms are implemented as functions that receive continuations as representations of blocks to execute. We also show how the envisioned language can control which continuations are passed and in which scope these continuations are executed.

We can conclude that Co-op improves the composition technology by providing a language that supports advanced composition. However, further development is needed to improve the maturity of the Co-op prototype.

Discussion

While our implementations of the Co-op language demonstrate that it is possible to make a language sufficiently powerful to define a wide range of composition mechanisms, some questions remain. For example, the execution performance of Co-op programs must be optimized to make this approach ready for practical use.

Optimizations are necessary because the dynamic evaluation of composition operators adds an overhead to each message send in the program. To compensate, our Co-op prototype already provides a high degree of declarativity in the definition of conditions, bindings and constraints. Thus, their effects on a message can be partly evaluated before runtime. However, it is still an open topic to actually implement such optimizations for Co-op and evaluate how much the runtime overhead can be reduced.

Whereas data composition is fully supported by our implementation, support for control-flow composition has only been designed. Thus, implementing our envisioned language for influencing control-flow composition is still future work.

10.2 RESOURCE-UTILIZATION MODELS

Resource optimization can also be carried out statically before the execution of software. Static optimization is usually achieved by modeling the resource consumption at the architectural level and performing analysis on the models. Such models are needed to understand the energy behavior of the system and to analyze which control strategy reduces the energy consumption. Therefore, we introduce the modeling notation *Resource-Utilization Models* that supports energy behavior and present guidelines for designing such models. To achieve resource-aware applications, we need to explicitly model (a) the components utilizing a resource of interest, (b) the relation between functional services and resource utilization of components, and (c) the interaction among components jointly utilizing a resource.

Besides identifying the necessity and usefulness of Resource-Utilization Models, we also developed design-level tools to automatically extract RUMs from existing source code. We explain why RUMs are over-abstractions and show how such over-abstraction can be extracted from source code by using the CEGAR method. Finally, we augment this extraction with profiling to automatically acquire energy information needed for the RUMs.

Discussion

Although we have shown how to extract RUMs from small software, extracting RUMs from large software systems remains challenging. Our tool chain also contains some manual steps, which should be automated to make extracting RUMs from large systems feasible.

Besides the CEGAR steps, also profiling is challenging. Therefore, related research on acquiring energy bounds through profiling should be included in our approach. Using static estimation methods can provide a faster and easier way to acquire counterexamples, so it is an additional future direction.

CO-OP/II PROTOTYPE

We have implemented a prototype of Co-op, named Co-op/II [P.1]. This prototype is implemented in Haskell [55, 78]. All example classes presented in chapters 3 and 4 have been tested using the prototype. Therefore, we can conclude that the presented language also works in practice.

The prototype provides a small set of built-in classes, namely: Null, Boolean, String, Array, Class, Constraint, Binding, and System. A Class has a method `getName()` to retrieve the class name and a method `new()` to create a new instance of the class, a Constraint and Binding have a method `activate()` and a method `deactivate()`. The lib directory contains the standard classes. It also contains interfaces, which have a purely descriptive use only. Interfaces cannot be loaded by the interpreter. The classes for which an interface definition is present are defined in Haskell code.

Debugging is supported in the prototype by adding the annotation `@Debug` to a message send. For every message send containing this annotation, the prototype will generate a diagram of the dispatch as shown in figure 47. These diagrams are similar to the diagrams we presented in chapter 3. Often, the generated diagrams are less clear because they are polluted with too much information and the layout is sub optimal. However, even though the diagrams are still a bit primitive, they provide quite valuable debugging information already. In the diagrams, the evaluation order is shown by dashed blue arrows and colors show the applicability of bindings.

The given example uses the class diagram of figure 48. The presented call is the following:

```
prof.@Debug makeAppointment("next Friday");
```

The variable `prof` contains an instance of `Professor`. We see that the call can be dispatched both to `StaffMember` and `Person`. However, the dispatch to `Person` is canceled by a skip constraint, resulting in the desired dispatch at `StaffMember`.

A.1 LIMITATIONS

The implemented prototype has some limitations. All core concepts are implemented to show that the language works, but some other features are left out. We will describe the limitations of the prototype in this section.

Built-in classes

Only a limited number of built-in data types is provided. The supported data types are Boolean, String, and Object. Arrays are available too, but their use is limited. Array creation is supported, but accessing elements is not possible, because integers are not available.

Figure 47: Debugging of a function call using inheritance

Figure 48: Class diagram of dispatch shown in figure 47

Arrays are used for passing explicit parameters to functions. When used in this way, the elements can be accessed because a name is assigned to each element.

Error reporting

Error reporting is limited. Some situations might give descriptive error messages, whereas others just return Haskell errors. Therefore, debugging can be difficult. The best way to handle errors is to check your code twice, especially for type errors and whether all variables are defined. If that is insufficient, you might have to use the Haskell debugger, which will not result in very user friendly debugging.

Single pass compilation

Our compiler runs a single pass only. Therefore, inner classes cannot be resolved from the outer class. This is not the desired behavior, but the workaround is easy: create an import statement for every inner class using its full qualified name. This import statement ensures that the outer class is aware of the existence of the inner classes.

Message properties

It should be possible to remove message properties. However, the prototype cannot handle undefined message properties properly. Therefore, it will return null whenever an undefined property is read.

A.2 PERFORMANCE

The prototype is designed to show the semantics of Co-op, not to be an efficient implementation of the Co-op runtime. Therefore, the prototype might use more memory and processing power than expected when executing any of the given examples. We have identified two main sources for high resource usage: memory management and constraint evaluation order. We will briefly explain these two deficiencies. Besides these deficiencies, we believe

many other optimization possibilities exist. Whether Co-op can achieve the same runtime performance as current mainstream programming languages is open for further research.

MEMORY MANAGEMENT The Co-op runtime does not have a garbage collector and does not provide the programmer with the ability to deallocate objects which are used no longer. Therefore, every object which is created once, will remain in memory forever, causing excessive memory consumption. The only exception is that the prototype finalizes messages when these are not used any more, which is important because bindings cause excessive message rewrites. However, this finalization does not apply to message properties. Thus, the application of bindings still causes an increase in memory usage.

CONSTRAINT EVALUATION ORDER Binding applicability and constraint evaluation are defined as two separate concerns. Therefore, they are implemented separately: binding applicability is calculated before constraints are evaluated. However, sometimes it is not necessary to know whether a certain binding is applicable or not, because the constraints prevent the execution of this binding anyway. Leaving the applicability of such a binding unevaluated might be a large performance gain, because the calculation of binding applicability can be lengthy due to its recursive definition.

An example of a function call where evaluating binding applicability is difficult, is a call using multiple inheritance. The dispatch of such a call is too difficult to show in detail here, but an overview is presented in figure 49. This diagram is extremely large because lookup is performed along every inheritance path until it reaches the topmost parent. However, when we zoom in on the root of the dispatch tree in figure 50, we see that the default binding succeeds. Therefore, the constraints cancel all other bindings which are present. Thus, it is unnecessary to evaluate the applicability of these other bindings. This means that evaluation of the root would be enough, which is considerably less than the evaluation of the full tree.

Figure 49: Difficult function call using multiple inheritance

Figure 50: Root of the dispatch tree shown in figure 49

B | COMPOSITION LIBRARY

It is possible to define many different composition operators in Co-op. In this appendix, we will explain three of these composition operators, to show how composition operators can be created. Further, we include the code of the composition operators that were mentioned in chapter 4.

B.1 DETAILS OF COMPOSITION OPERATORS

B.1.1 Static

Static method calls are not a built-in concept of Co-op. However, it is easy to write a static method call, because Co-op will load the class `Static` listed in appendix B.2.1 before executing any program. This class defines the composition operator for static method calls, which will be explained in this section. Some relevant code snippets from appendix B.2.1 are also listed in this section. When we do so, we use the line numbers from the full listing, so it is easy to locate the same code in the appendix.

Consider that we have the following class, defining a single method, `someMethod`, which behaves like a static method:

```
1  class SomeClass {  
2      method @ImplicitParameters([]) someMethod() {  
3          // Some method body  
4      }  
5  }
```

Now we can execute `someMethod` in the following way:

```
SomeClass.someMethod();
```

This call will result in the message send shown in figure 51 (irrelevant message properties are omitted). The steps in the dispatch are explained below.

The original call has target `SomeClass`, which is an instance of `coop.lang.Class` representing `SomeClass`. Therefore, the default binding will try to locate an implementation of the method `someMethod` in `coop.lang.Class`, which fails. Thus, the default binding fails. The class `Static` defines a single binding:

```
9  binding staticBinding = (targetType == Class & messageKind == "Call" & target != null) {  
10      targetType = target;  
11      target = null;  
12      remove this;  
13  }
```

Figure 51: Static dispatch example for the class shown in appendix B.2.1

This staticBinding matches the first message and rewrites it to the second one shown in figure 51. We see that the target type has changed to SomeClass, which contains an implementation for someMethod. Therefore, the default binding succeeds here.

We also see that a skip constraint is present between the default and static binding:

```
15 constraint defaultBeforeStatic = skip(System.defaultBinding, staticBinding);
```

In this example, the constraint does not do anything, because the default binding fails. We can see the working of the constraint when we create a new instance of SomeClass:

```
SomeClass.new();
```

The method new is implemented in coop.lang.Class, so the default binding succeeds. Because the default binding is successful, the static binding is skipped. Therefore, the evaluation of the static binding can be omitted, as shown in figure 52.

B.1.2 Event Notification

It can be desired to create an event which does not necessarily have to be handled. For example, the message initializeClass, which is implicitly called during class loading, may be ignored if a class does not require initialization. In order to achieve this behavior, Co-op loads the class shown in appendix B.2.2 by default. This class defines a single binding and constraint and the method noDispatch used by the binding:

```
6 binding implementationOptional = (messageKind == "Call" & message @==
  @ImplementationOptional) {
7   targetType = DispatchOptional;
```

Figure 52: Construction of an instance of SomeClass

```

8   name = "noDispatch";
9   parameters = [];
10  message @- = @ImplementationOptional;
11  }
12
13  constraint whenDefaultFails = skip(System.defaultBinding, implementationOptional);
14
15  method @ImplicitParameters([]) noDispatch() {
16    // Nothing to be done when no dispatch is required
17  }

```

Also, any call to `initializeClass` is annotated with `@ImplementationOptional`:

```
SomeClass.@ImplementationOptional initializeClass();
```

When a call is annotated with `@ImplementationOptional`, the dispatch will look as shown in figure 53. Thus, if the default binding fails, the `implementationOptional` binding will call the empty method `noDispatch`. Therefore, the call succeeds and nothing happens (unless some other dispatch handles this call, of course).

B.1.3 Multiple Inheritance

A more interesting example is multiple inheritance. As shown in section 4.4, many types of inheritance exist. In this section, we will show an excerpt of our inheritance implementation, which is the part that models class-based, delegation based, asymmetric descendant-driven multiple inheritance only.

Both method and field lookup are influenced by inheritance. The influence on method lookup will be explained in the next subsection, the influence on field lookup in the subsection thereafter.

Method Inheritance

The influence of inheritance on method lookup is defined by the class `MethodInheritance`. This class specifies multiple types of inheritance, of which we explain multiple inheritance in this section. The code defining multiple inheritance is fully listed in appendix B.2.3.

Figure 53: Dispatch when a call is annotated with @ImplementationOptional

Figure 54: Staff member class diagram

CREATING INHERITANCE RELATIONS The class `MethodInheritance` specifies the primitives for multiple types of inheritance. Multiple inheritance can be defined using these primitives as follows:

```

class MultipleInheritance {
    // Call this method for every parent type
    method @ImplicitParameters([]) subclassOf(childType, parentType) {
        MethodInheritance.inheritsAsymmetricDescendantDriven(childType, parentType);
        FieldInheritance.subclassOf(childType, parentType, InheritedFieldAccess);
    }
}

```

The class is a factory to create multiple inheritance easily. It allows using multiple inheritance in the following way:

```

MultipleInheritance.subclassOf(StaffMember, Person);
MultipleInheritance.subclassOf(StaffMember, Cost);

```

These two lines of code specify that `StaffMember` inherits from both `Person` and `Cost`, which is also shown in figure 54. For each inheritance relation between two classes, an instance of `MethodInheritance` is created using this call. This instance stores the child and parent type in the following variables:

```

10  var childType;
11  var parentType;

```

VIRTUAL METHOD CALLS The following binding specifies virtual method lookup:

```

14  binding virtualBinding = (messageKind == "Call" & targetType == this.childType &
    message @!= @Final) {
15      targetType = this.parentType;
16  }

```

When the lookup type of a message is equal to the child type of an inheritance relation, and the call is not annotated with `@Final`, the method is also invoked on its parent class by this `virtualBinding`. This means that two different bindings match the same message: the default binding and the `virtualBinding`. This would lead to the execution of two methods when both the parent and child type implement a method with the specified name, which is the composing vertical combination explained in section 4.2. However, we would like to have asymmetric vertical combination. Therefore, we add a constraint between the default binding and the `virtualBinding`:

```

41  constraint asymmetricDescendantDriven = skip(System.defaultBinding, virtualBinding);

```

This constraint specifies that if the default binding succeeds, any `virtualBinding` is skipped. This way, method definitions in the child type effectively override those in the parent type, while methods not defined in the child type are forwarded to the parent type, as intended by asymmetric descendant-driven inheritance.

Note that the dispatch might be a multi-stage process: if the parent type is involved in another inheritance relation, the rewritten message may be rewritten again. These rewrites continue until the message reaches an ancestor implementing the method or until the root of the inheritance DAG is reached and the message send fails.

SUPER METHOD CALLS Many object-oriented languages provide special keywords to refer to related classes in the program structure. For example, in Java the keyword `super` can be used by subclasses to call method implementations defined in an ancestor class, even if the subclass overrides the original method implementation. However, the keyword `super` does not refer to an object in the same sense as this can be considered a normal object. Keywords such as `super` are rather an indication to the message dispatch mechanism of a language that it should deviate from its normal procedure: instead, the lookup procedure followed to dispatch the message should be started at the parent of the class in which the keyword occurs.

In Co-op, annotations are used to influence dispatch. Therefore, instead of introducing a keyword such as `super`, we model the desired behavior using the call annotation `@Super`. For example, this annotation can be used as follows:

```

this.@Super makeAppointment(date);

```

The `superBinding` implements this behavior:

```

21  binding superBinding = (messageKind == "Call" & message @== @Super & targetType
    == this.childType & (message.annotations.get("inheritance.Super").value == null |
    message.annotations.get("inheritance.Super").value == this.parentType.getName())) {

```

```

22     targetType = this.parentType;
23     message @- = @Super;
24 }

```

This condition also reads the value of the super annotation, which can be used during multiple inheritance to specify which parent should be used. When @Super is specified, the call is dispatched to all parents. However, by specifying @Super("Person"), the call would only be dispatched to the parent of type Person, if such a parent exists.

We see that the search for an implementation starts at the parent of the target (the target is this, so it equals the sender). Also, the annotation @Super is removed, so the search proceeds as a virtual method call described before.

Again, it is possible that both the default and superBinding match. Here, it is clear that only the superBinding should be executed, which is specified by the following constraint:

```

46 constraint superSkipsDefault = skip(superBinding, System.defaultBinding);

```

Field Inheritance

The influence of inheritance on field lookup is similar to the influence on method lookup. The difference is that for field lookup an instance is required, whereas for method lookup knowing the type defining a method is sufficient. The implementation of field inheritance is shown in appendix [B.2.4](#).

SUPER CONSTRUCTOR CALLS When an instance of a child class is created, it is often desired that the constructor of the super class is called. In Co-op, inheritance is realized by delegation, which means that the fields of ancestors are not stored in the child itself. Therefore, it is really necessary to create its ancestors when a child is constructed. This is done by creating the parent after the child has been created.

```

12 binding superConstructorBinding = (messageKind == "Call" & name == "new" & parameters
    == [] & target == this.childType) {
13     targetType = FieldInheritance;
14     target = null;
15     name = "createParent";
16     parameters = [result, this.parentType]; // The newly created object is "result"
17     this = this;
18 }

```

The superConstructorBinding, shown above, defines calling the super constructor when the child is constructed. This is achieved by calling the method createParent, shown below, with as arguments the newly created child—the result of the constructor called before—and the type of the parent. The method createParent creates a field inheritance relation between the child and a newly constructed parent instance. This relation ensures that fields of any ancestor are accessible through the child. Further, createParent returns the child again, ensuring that the original constructor call returns the constructed object.

```

28 method createParent(child, parentType) {
29     var access = this.accessType.new();

```

Figure 55: Class and object diagram of a professor

```

30  access.initDispatch(child, parentType.new());
31  return child;
32  }

```

We see that the `superConstructorBinding` uses the result of the original constructor. Of course, this result is only available after the original constructor executed successfully. This execution requirement is specified using two constraints:

```

20  constraint superConstructorConstraint = cond(System.defaultBinding,
        superConstructorBinding);
21  constraint superConstructorOrderConstraint = pre(System.defaultBinding,
        superConstructorBinding);

```

Using the described creation strategy, the inheritance structure shown in figure 55a will create the object structure shown in figure 55b. An instance of every ancestor is created. These ancestors are linked to the child by instances of `FieldInheritance`. The creation of all these objects is transparent to the programmer, because only a reference to the child `prof` is returned. The other objects are not directly accessible.

B.2 CODE OF COMPOSITION OPERATORS

B.2.1 Static Method Calls

```

1 package coop.lang;
2
3 import coop.lang.Class;
4 import coop.lang.System;
5
6 class Static {
7   // Binding for static method lookup, which is applicable only if the default binding fails
8   // The check for a null target avoids infinite resends, Class Class will resend to itself only once
9   binding staticBinding = (targetType == Class & messageKind == "Call" & target != null) {
10     targetType = target;
11     target = null;
12     remove this;
13   }
14
15   constraint defaultBeforeStatic = skip(System.defaultBinding, staticBinding);
16
17   method initialize() {
18     this.defaultBeforeStatic().activate();
19     this.staticBinding().activate();
20   }
21
22   method @ImplicitParameters([]) initializeClass() {
23     Static.new().initialize();
24   }
25 }

```

Listing 18: Dispatch for static method calls

B.2.2 Dispatch Optional

```

1 package coop.lang;
2
3 import coop.lang.System;
4
5 class DispatchOptional {
6   binding implementationOptional = (messageKind == "Call" & message @== @ImplementationOptional)
7     {
8     targetType = DispatchOptional;
9     name = "noDispatch";
10    parameters = [];
11    message @- = @ImplementationOptional;
12  }
13
14  constraint whenDefaultFails = skip(System.defaultBinding, implementationOptional);
15
16  method @ImplicitParameters([]) noDispatch() {
17    // Nothing to be done when no dispatch is required
18  }
19
20  // Load this class before any other class.
21  // This class has a static initializer and after it has run, classes don't require one.
22  method @ImplicitParameters([]) initializeClass() {
23    var opt = DispatchOptional.new();
24    opt.whenDefaultFails().activate();
25    opt.implementationOptional().activate();
26  }
27 }

```

Listing 19: Dispatch optional

B.2.3 Method Inheritance

```

1 package inheritance;
2
3 import coop.lang.System;
4 import coop.lang.ImplicitParameters;
5 import inheritance.Super;
6 import inheritance.Inner;
7 import inheritance.Final;
8
9 class MethodInheritance {
10     var childType;
11     var parentType;
12
13     // Binding for virtual method lookup, which is applicable only if the default binding fails
14     binding virtualBinding = (messageKind == "Call" & targetType == this.childType & message @!=
15         @Final) {
16         targetType = this.parentType;
17     }
18
19     // Binding for explicit super calls, causing the default binding to become inapplicable.
20     // Note that the targetType is not checked against the senderType, so besides this.@Super,
21     something.@Super will also work.
22     // We use either all super classes or the chosen one.
23     binding superBinding = (messageKind == "Call" & message @== @Super & targetType ==
24         this.childType & (message.annotations.get("inheritance.Super").value == null |
25         message.annotations.get("inheritance.Super").value == this.parentType.getName())) {
26         targetType = this.parentType;
27         message @- = @Super;
28     }
29
30     // When @Inner is specified, the call is dispatched at the topmost class under senderType.
31     binding innerBinding = (messageKind == "Call" & message @== @Inner & targetType ==
32         this.childType & senderType == this.parentType) {
33         message @- = @Inner;
34         message @+ = @Final;
35     }
36
37     // End inner call at child, because it is the last class in the hierarchy (will be canceled if not)
38     binding innerBindingLast = (messageKind == "Call" & message @== @Inner & targetType ==
39         senderType) {
40         targetType = MethodInheritance;
41         target = null;
42         name = "noop";
43         message @- = @Inner;
44     }
45
46     // Constraints to specify various kinds of inheritance
47     constraint asymmetricDescendantDriven = skip(System.defaultBinding, virtualBinding);
48     constraint asymmetricParentDriven = skip(virtualBinding, System.defaultBinding);
49     constraint composingDescendantDriven = pre(System.defaultBinding, virtualBinding);
50     constraint composingParentDriven = pre(virtualBinding, System.defaultBinding);
51
52     constraint superSkipsDefault = skip(superBinding, System.defaultBinding);

```

```

48 constraint innerSkipsDefault = skip(innerBindingLast, System.defaultBinding);
49 constraint innerConstraint = skip(innerBinding, virtualBinding);
50 constraint innerLastConstraint = skip(innerBindingLast, virtualBinding);
51
52 method initDispatch(childType, parentType) {
53     this.childType = childType;
54     this.parentType = parentType;
55     // Activate all constraints
56     this.superSkipsDefault().activate();
57     this.innerConstraint().activate();
58     // Activate all bindings
59     this.virtualBinding().activate();
60     this.superBinding().activate();
61     this.innerBinding().activate();
62 }
63
64 method @ImplicitParameters([]) inheritsAsymmetricDescendantDriven(childType, parentType) {
65     var inh = MethodInheritance.new();
66     inh.asymmetricDescendantDriven().activate();
67     inh.initDispatch(childType, parentType);
68 }
69
70 method @ImplicitParameters([]) inheritsAsymmetricParentDriven(childType, parentType) {
71     var inh = MethodInheritance.new();
72     inh.asymmetricParentDriven().activate();
73     inh.initDispatch(childType, parentType);
74 }
75
76 method @ImplicitParameters([]) inheritsComposingDescendantDriven(childType, parentType) {
77     var inh = MethodInheritance.new();
78     inh.composingDescendantDriven().activate();
79     inh.initDispatch(childType, parentType);
80 }
81
82 method @ImplicitParameters([]) inheritsComposingParentDriven(childType, parentType) {
83     var inh = MethodInheritance.new();
84     inh.composingParentDriven().activate();
85     inh.initDispatch(childType, parentType);
86 }
87
88 method @ImplicitParameters([]) initializeClass() {
89     var inh = MethodInheritance.new();
90     // Activate all static constraints
91     inh.innerSkipsDefault().activate();
92     inh.innerLastConstraint().activate();
93     // Activate all static bindings
94     inh.innerBindingLast().activate();
95 }
96
97 method @ImplicitParameters([]) noop() {
98 }
99 }

```

Listing 20: Method inheritance

B.2.4 Field Inheritance

```

1 package inheritance;
2
3 import coop.lang.System;
4 import coop.lang.ImplicitParameters;
5
6 class FieldInheritance {
7     var childType;
8     var parentType;
9     var accessType;
10
11     // Call the super constructor to create a parent
12     binding superConstructorBinding = (messageKind == "Call" & name == "new" & parameters == [] &
13         target == this.childType) {
14         targetType = FieldInheritance;
15         target = null;
16         name = "createParent";
17         parameters = [result, this.parentType]; // The newly created object is "result"
18         this = this;
19     }
20
21     constraint superConstructorConstraint = cond(System.defaultBinding, superConstructorBinding);
22     constraint superConstructorOrderConstraint = pre(System.defaultBinding, superConstructorBinding);
23
24     /**
25     * Call the super constructor to create an instance of the super object,
26     * create a relation between the super object and the this object, and
27     * return the this object.
28     */
29     method createParent(child, parentType) {
30         var access = this.accessType.new();
31         access.initDispatch(child, parentType.new());
32         return child;
33     }
34
35     method initDispatch(childType, parentType, accessType) {
36         this.childType = childType;
37         this.parentType = parentType;
38         this.accessType = accessType;
39         // Activate all constraints
40         this.superConstructorConstraint().activate();
41         this.superConstructorOrderConstraint().activate();
42         // Activate all bindings
43         this.superConstructorBinding().activate();
44     }
45
46     method @ImplicitParameters([]) subclassOf(childType, parentType, accessType) {
47         FieldInheritance.new().initDispatch(childType, parentType, accessType);
48     }
49 }

```

Listing 21: Field inheritance

C | CO-OP/III

To improve the usability of Co-op and to address limitations in Co-op/II, we have been working on an improved prototype, compared to Co-op/II (which is described in appendix A). This latest prototype is named Co-op/III and has only been implemented partly. Its implementation is available on the Co-op website¹

The main goal of Co-op/III is to improve the usability of Co-op, which leads to the following improvements:

IDE Co-op/III is developed as an Eclipse² plugin, which provides syntax coloring and error highlighting.

JAVA INTEGRATION Co-op/III integrated with Java, so a proper standard library is available.

COOPUNIT Co-op/III provides CoopUnit for unit testing by directly integrating with JUnit, so writing unit tests is as easy as in Java.

PERFORMANCE IMPROVEMENTS Co-op/III compiles to JVM bytecode and, therefore, benefits from all optimizations performed by the JVM, so we expect applications to run much faster than in Co-op/II, which is not optimized for speed at all.

Besides these improvements, we also plan to address some limitations of Co-op/II:

ANNOTATIONS Co-op/II has only limited support for annotations, but we show in chapter 4 that some constructs can be expressed more naturally when annotations are fully supported. Therefore, Co-op/III aims to have full annotation support.

CONTROL-FLOW COMPOSITION Chapter 5 outlines advanced control-flow composition that cannot be realized in Co-op/II, so our aim is to support such composition in Co-op/III.

The foundations of Co-op/III are implemented by Havva Gülay Gürbüz during her internship at the University of Twente. She realized all four usability improvements, but did not implement the evaluation of bindings and constraints. We will provide implementation details in the next section and described the unimplemented features on which we worked in the section thereafter.

¹ A description how Co-op/III can be installed in an Eclipse IDE is provided in the wiki of the SourceForge project *Co-op* at co-op.sf.net.

² The Eclipse IDE is available at www.eclipse.org.

C.1 IMPLEMENTATION

Co-op/III is implemented with Xtext³ as a plugin for the Eclipse IDE². This section provides some technical details of the prototype and gives a few listings, which have been tested with the Co-op/III prototype.

c.1.1 Code Generation

For the ease of code generation and debugging, we compile Co-op/III code to Java source code. Compiling to Java bytecode would provide more expressiveness, but would also increase the complexity of code generation. Since most Co-op expressions can be mapped to Java source code easily, we decided that compiling to source code is the best solution as long as it does not limit our possibilities.

Because Java is typed statically, the generated code should be correct under static type checking. Therefore, we generate a class `CoopObject`, which is the super class of any object defined in Co-op. That is, the relation between the class `CoopObject` and any class defined in Co-op is the same as the relation between the class `Object` and any class defined in Java. `CoopObject` implements every method that is used somewhere in the Co-op code, in order to type check correctly. These methods all throw a runtime exception, since only when an actual implementation is provided in Co-op, a call to them can succeed. The downside of this method is that incremental compilation is not supported; all Co-op classes must be compiled together.

Method Calls

In Co-op, method calls can be annotated, which is not possible in Java. Therefore, we have to keep our own stack storing these annotations. This is done by creating a helper method for every method call inside Co-op. This helper method (a) pushes all annotations on the stack, (b) performs the actual call, and (c) pops the annotations. This way, the annotations belonging to the current call are always available.

Field Access

In Co-op, field accesses are message sends just like method calls. However, we cannot define all fields inside `CoopObject` like we do with methods, and override them in subclasses, since field lookup cannot be dynamic (field lookup is always performed at compile time). Therefore, fields could be defined in one of the following three ways:

1. Define all fields F in `CoopObject` and define a subset S of these in a subclass, as we do for methods. In this case, the subclass will have all fields $F + S$, where for all fields in S two instances with the same name exist (the one from S and the one from F). This has the following disadvantages:
 - It can cause unexpected behavior (depending on the static type of a variable, one of the fields is used).

³ Information on the Xtext framework can be found at eclipse.org/Xtext/.

- It increases the memory consumption (all fields $F + S$ are stored for each object, whereas only fields S are used).
 - It increases the complexity of debugging (how to distinguish between multiple fields with the same name).
2. Define all fields in `CoopObject` and do not define a subset of these in the subclasses. The disadvantage here is that we can no longer identify at runtime which class contains which fields, since all fields are always available, which can result in unexpected behavior at runtime.
 3. Generate a getter and a setter for each field and use these to access the fields. Since field access in Co-op is modeled the same as method calls, changing field access to method calls does not have any adverse effects. When the getter and setter methods are left unimplemented in `CoopObject` (that is, an implementation that always throws a runtime exception is provided), we do not have to define the fields in `CoopObject`. Fields are defined in the classes for which the Co-op code actually defines them, which ensures that we always know which class contains which fields, and only used fields are stored in memory.

Solution 3 is implemented in the actual prototype.

Naming Convention of Generated Methods

Methods which are defined in Co-op, in general, retain their defined name when compiled to Java, except the following ones:

- Method names that are Java keywords cannot be used inside Java. In this case, a `$` is postfixed to the method name.
- A few methods that are defined in `java.lang.Object` are treated as if these names are Java keywords (described in the following paragraph).
- Static methods are prefixed with a `$` to avoid hiding instance methods, which is not allowed in Java.
- Special test methods are generated allowing JUnit to invoke Co-op code (these methods are not used by Co-op itself) for any method annotated with the annotation `@Test`.

For fields, we use a similar naming scheme. The mapping of names is performed by the compiler and, therefore, is transparent to the programmer. No programmer will see the methods with a `$` in their name, except when debugging Co-op code inside a Java debugger.

METHODS OF `JAVA.LANG.OBJECT` Co-op does not contain any inheritance, therefore an object should not have a basic set of methods. However, we cannot avoid classes to inherit from `java.lang.Object`, which gives them a few methods (e.g., `hashCode()` and `toString()`). Since their signatures do not match the signatures of Co-op methods, they cannot be used or overridden from Co-op. Therefore, we have decided to hide these methods from Co-op in the following way:


```

1 package hello;
2
3 import net.sf.co_op.coopiii.lang.System;
4
5 class Hello {
6     method sayHello(var name){
7         return "Hello " + name;
8     }
9
10    method main(var args){
11        var hello;
12        hello = Hello.new();
13        System.println(hello.sayHello("world"));
14    }
15 }

```

Listing 22: Hello world in Co-op/III

- If a Co-op class defines a method called `hashCode()`, it is compile to `hashCode$()`, so it does not interfere with the method defined in `Object`.
- When a Co-op class calls `hashCode()`, it actually calls `hashCode$()`, which is the version defined by Co-op. Only when the target is a Java class instead of a Co-op one, the Java integration will ensure that the implementation of `java.lang.Object` is called and the result is wrapped to become a `CoopObject`.
- If Java code calls `hashCode()` on any Co-op object, any implementation of `hashCode()` in Co-op is not visible (since it was compiled to `hashCode$()`).

c.1.2 CoopUnit

Unit testing is important for any piece of software, including ours. Therefore, JUnit integration is added to Co-op, so unit tests can be written easily. For example, the method `sayHello` from the program shown in listing 22 can be tested as shown in listing 23. An example of the output generated by JUnit is shown in figure 56.

c.1.3 Annotations

We would like to represent annotations in multiple situations:

1. At definition time on methods, fields, etc.
 - a) Java annotations can be used, but such annotations require literals as parameters, whereas we would like the freedom to use any Co-op type.
 - b) Only Java annotations can be stored here, so a custom representation is only possible if it can be mapped onto Java annotations.

Figure 56: Example of testing Co-op/III with JUnit

```

1 package hello;
2
3 import hello.Hello;
4 import org.junit.Assert;
5 import net.sf.co_op.coopiii.lang.test.Test;
6
7 class HelloTest {
8     method @Test() testHello(){
9         var hello;
10        hello = Hello.new();
11        Assert.assertEquals("Hello Havva", hello.sayHello("Havva"));
12    }
13 }

```

Listing 23: Unit test in Co-op/III

2. At call side.

- a) Java annotations can be used, but it requires the instantiation of proxies implementing these annotations and has the same limitations as item 1a.
- b) We can use our own implementation, not conforming to Java annotations at all, which gives us full freedom here, but these do not work at definition time as described at item 1.

Possible solutions are:

1. Using items 1a and 2a together, and encoding all Co-op objects as integer references into Java annotations would work. However, this still has limitations:
 - a) Co-op objects used by annotations must be stored in a central store.
 - b) Accessing the parameter values of annotations requires indirection, since these are in the central store.
 - c) Using Java annotations inside Co-op is not possible since the parameters values should not be references to Co-op objects here.
2. Using item 2a and put all annotations in a custom store (so they are not present at definition time as in Java).
 - a) As in the previous solution, this also requires a special store.
 - b) Since we cannot reuse Java's reflective capabilities, the store itself will provide all reflective capabilities. This will avoid the need for any indirection when accessing annotation parameters.
 - c) Java annotations cannot be fully used inside Co-op, as in the previous solution.

We have chosen solution 2, since it provides the largest flexibility. Integration with Java annotations can be achieved in future by creating a way to map Co-op annotations to their

```

1 annotation A {
2     method value() default @A() 0;
3 }

```

Listing 24: Cyclic annotation A

Java counterparts, if we see this as a necessity for using important Java libraries. Full integration does not seem to be feasible, since we do not want to compromise the flexibility of the Co-op model.

We have a working example of adding annotations, but reflection has not been designed and tested yet.

Annotations can be specified on the default value of an annotation property, but this easily creates cyclic references, which make the annotation unusable. Therefore, it is better to avoid adding annotations to default values. For example, when we use annotation `@A` on the default value of `A.value` (shown in listing 24), then in order to instantiate `@A`, the default value must be calculated (this is currently done in any case, even if the instance that is being created overrides the default value), which requires `@A` to be instantiated. This cyclic dependency will result in a runtime error. In future we could change this behavior to one of the following:

- Leave it as is and accept runtime errors.
- Try to limit the number of runtime errors, e.g., by only calculating the default value when needed. This cannot avoid all errors.
- Do not allow the use of annotations on default values. Right now, default values are very expressive, they can be any expression. Why would we need such expressiveness? It might be better to limit the expressiveness of default values and avoid runtime errors caused by default values.
- Try to detect cyclic dependencies and disallow these, but keep allowing all other situations. Probably, this cannot be sound and complete due to undecidability, so a compromise must be made.

C.1.4 Java Integration

The integration of Co-op with Java requires the following situations to be handled:

1. Co-op objects should be able to live in the Java world.
2. Java objects should be able to live in the Co-op world.
3. Co-op code should be able to call methods of Java objects.

Figure 57: Java integration

Co-op Objects in the Java World

Since Co-op objects all inherit from the Java class `Object`, Co-op objects can easily be passed where Java expects an `Object`. Whenever Java expects another type `T` there are two possibilities:

1. The object passed is a Java object of type `T` living in the Co-op world, so it can easily be passed to the Java world.
2. The object passed is a Co-op object conforming to `T`. For now, we cannot check the conformance so we do not allow this.

Java Objects in the Co-op World

To allow Java objects to live in the Co-op world, we can wrap them inside a wrapper class. Since Co-op is typed dynamically, a generic wrapper class `JavaObject` is defined. Whenever a Java object enters the Co-op world, it is wrapped inside a `JavaObject` and when it wants to enter the Java world again, the only thing to do is unwrap it before letting it in.

Calling Methods on Java Objects

Since Java resolves methods at compile time using their static types, Co-op cannot provide the same behavior. Co-op will provide dynamic dispatch for these objects. We decided to use asymmetric dynamic dispatch for any call made from Co-op to Java. We expect that this will induce the behavior expected by a Java programmer in most cases. We also plan to provide some annotations to allow static method selection.

The current prototype has an experimental implementation of Java integration. This implementation causes some additional indirection that can be seen at binding level. Instead of the Java type, we always see the type `JavaObject` from the Co-op world, as shown in figure 57.

Currently, the left message is generated and then `JavaObject` calls the actual implementation in `String`. It would be desired to create the right message, but that requires a special binding to handle this message, which is not possible with the current state of the implementation. When we remove this additional indirection, we can remove all Co-op primitives

and replace them with Java classes. Co-op primitives provide more functionality than their Java counterparts, but a binding could add this functionality to the Java counterparts.

C.2 LIMITATIONS

The Co-op/III prototype has only been implemented partly, so it cannot run as much code as the Co-op/II prototype. This section will outline the main limitations.

C.2.1 Bindings and Constraints

We envisioned to use an advanced dispatching framework, such as ALIA4J [12], for implementing the bindings and constraints of Co-op. The idea is that an advanced dispatching framework simplifies the implementation of bindings and constraints, while also providing free performance improvements. That is, the work that has been invested in optimizing the performance of the advanced dispatching framework can be reused without investing a lot of time on performance optimization.

Van Hemert [T.1] investigated how Co-op bindings and constraints can be implemented using the framework ALIA4J and how such an implementation compares to a pure Java implementation. Whereas the primitives provided by ALIA4J cover the advanced dispatch used in many aspect-oriented languages, they do not precisely cover the primitives required by Co-op. Therefore, Van Hemert concludes that the advantages of using ALIA4J do not outweigh the disadvantages, so an implementation in pure Java is the better choice. This means that we cannot reuse the performance optimization that is part of ALIA4J, which reduces the performance benefits that Co-op/III can provide over Co-op/II. Van Hemert outlines the basics for implementing bindings and constraints in pure Java, but his work has not been integrated into Co-op/III.

C.2.2 Control-Flow Composition

The control-flow composition that is outlined in chapter 5 has not been tested in an actual prototype. Laarakkers [T.2] planned to change the output of the Co-op compiler to continuation-passing style (CPS), without changing the syntax of Co-op itself. This would provide the flexibility needed to achieve control-flow composition. Scala [71] is a language that is compatible with Java and that provides delimited continuations. Therefore, compiling to Scala instead of Java seems a good choice. Laarakkers concludes that closures are enough to compile to CPS; the delimited continuations are not needed. Thus, when adding CPS to the compiler, the target does not have to be changed to Scala, it can remain Java source code. Unfortunately, Laarakkers only provides a partial implementation of compilation to CPS, so control-flow composition cannot be achieved yet.

c.2.3 Implicit Parameters

Implicit parameters are parameters that are passed to a method implicitly. In Co-op/II, a method must declare all implicit parameters that it uses. If methods would not declare all implicit parameters, the following situations would fail at runtime due to accessing undefined variables:

1. Typos in variable names.
2. Bindings that unexpectedly leave variables undefined.

Whereas Co-op/II tries to avoid these runtime errors, we reconsider this behavior for Co-op/III, because of the following reasons:

1. Typos in method names are not discovered at compile time either, so not declaring implicit parameters brings their semantics closer to that of methods.
2. If implicit parameters are declared, bindings that unexpectedly leave variables undefined will fail during dispatch, otherwise they will fail during execution (e.g., calling a non-static method in a static way will then cause failure when “this” is referenced for the first time, whereas now the dispatch will fail). Both cases are similar runtime errors.

The situation described in item 2 is that methods fail somewhere *during* execution, instead of *before* execution. We consider it beneficial that execution fails as early as possible, which can be achieved by enforcing that all implicit parameters that are used in a method are defined, before executing any code. Such enforcement can be performed easily by adding a single statement for each used implicit parameter to the start of the method. For example, for “this” the following statement would be added:

```
CoopObject this = CoopRuntime.getImplicitParameterValue("this");
```

We decided that this would be the choice for Co-op/III, since it provides nearly the same semantics as used in Co-op/II, reduces the needed syntax, and increases the similarity between method and implicit parameter lookup. However, implicit parameters have not been implemented yet in Co-op/III.

D

MEDIA PLAYER EXAMPLE

In section 7.4 we analyze the energy consumption of a media player. The energy consumption is analyzed while three different controllers are used. These three controllers are shown in figure 58. For each controller, a distinct analysis is performed with the given scenario. The controller is instantiated in the following way:

```
Control = Controller();
```

Replacing this statement with the following one changes the controller:

```
Control = ControllerBurst(app_buffer);
```

Shared variables, such as `app_buffer`, are passed to the template during instantiation.

We also have an instance of the user behavior shown in figure 30 on page 89. When we provide UPPAAL a query such as `E<> Scenario.done`, UPPAAL will provide us with an example that we can use to analyze the energy consumption during the scenario (as shown in figure 59).

The network manager that we use is based on the description given by Qian et al. [81]. Figure 59 shows that the energy consumption differs when the characteristics of the network are different. The network that we use is summarized in table 10. For the analysis, we vary both the quality of the music (32 or 64 kbps) and the network speed in the FACH state (32 or 48 kbps). We see that in every analysis the fast dormancy performs best, but downloading in bursts is not efficient if it causes regular switching to the DCH state, which would not be needed otherwise.

Inactivity timer	
DCH → FACH	6 s
FACH → IDLE	4 s
Promotion time	
FACH → DCH	1.3 s
State radio power	
IDLE	0 mW
FACH	400 mW
DCH	600 mW
Promotion radio power	
FACH → DCH	480 mW
Bandwidth	
IDLE	0 kbps
FACH	32 or 48 kbps
DCH	144 kbps

Table 10: Parameters of the network

(a) Basic controller

(b) Controller that downloads in bursts

(c) Controller that downloads in bursts and offers fast dormancy

Figure 58: Various controllers

(a) Music with a bitrate of 64 kbps, network with 48 kbps speed in FACH state

(b) Music with a bitrate of 32 kbps, network with 48 kbps speed in FACH state

(c) Music with a bitrate of 32 kbps, network with 32 kbps speed in FACH state

Figure 59: Analysis of scenario with various network characteristics

E | MAGIC RUM

The tool MAGIC RUM converts the output of MAGIC, which is in DOT format, to a UPPAAL XML file. MAGIC RUM is written in Python 2.7 and has the following dependencies:

- **pygraphviz**, installed in Python
- **pyparsing**, installed in Python
- **tarjan**, installed in Python
- **cvc3** or **cvc4**, runnable from the command line

E.1 USING MAGIC RUM

To run MAGIC RUM, the method *magic_rum* must be called. This method requires five arguments to configure the process, an example configuration can be seen in the script *magic_rum-NetworkManager_key3.py* shown in listing 25. The parameters are as follows:

- **file**: the base filename for the input graph (e.g.: use *NetworkManager_key3* when the input file is called *NetworkManager_key3.dot*).
- **relevant primitives**: all variables that are relevant to keep, plus the special primitive *INIT*.
- **relevant operations**: all operations that are relevant to keep.
- **variables**: the typed declarations of all relevant primitives and operations as used by UPPAAL (operations have the type *chan*).
- **cvc variables**: the typed declarations of all relevant primitives as used by the CVC theorem prover.

E.2 THE SIMPLIFICATION PROCESS

The conceptual steps of the simplification are explained in detail in section 8.3.1; this are steps 3, 4 and 6 of the simplification process, which we describe in this chapter. The method *magic_rum* shown in listing 26 performs the simplification process as follows:

1. Extract the LTS after predicate abstraction from the output of MAGIC, because the output also contains intermediate graphs.

```

1 from magic_rum import magic_rum
2
3 file = 'NetworkManager_key3'
4 relevant_primitives = frozenset([ 'INIT', 'energy', 't1' ])
5 relevant_operations = frozenset([ 'DOWNLOAD_REQUEST', 'UPLOAD_REQUEST', 'MOVE',
6     'SEND', 'RECEIVE' ])
7 variables = {
8     'int' : ['inactivity', 'energy', 't1'],
9     'bool' : ['P0::temp_var_14'],
10    'chan' : ['SEND', 'RECEIVE']
11 }
12 cvc_variables = ""
13     inactivity, energy, t1 : INT;
14     P0::temp_var_14 : BOOLEAN;
15 ""
16 magic_rum(file, relevant_primitives, relevant_operations, variables, cvc_variables)

```

Listing 25: magic_rum-NetworkManager_key3.py

2. Remove labels from edges.
3. Remove unimportant predicates (section 8.3.1 describes how we decide which predicates are important).
4. Simplify predicates (appendix E.3 describes the details of predicate simplification, section 8.3.1 shows an example).
5. Move operations to separate states. Moving every operation to its own state allows us to easily add operations to edges later, while still keeping the simplification process easy by having no labels on edges yet.
6. Calculate the stutter bisimulation quotient.
7. Move operations to edges.
8. Serialize the model as a UPPAAL NTA.

E.3 PREDICATE SIMPLIFICATION

MAGIC adds to each state a set of predicates (and operations) that all hold in that state. To detect more states as equivalent, we perform the following steps:

1. Eliminate conjunctions: “A && B” is transformed to: “A”, “B”.
2. Remove irrelevant predicates: all predicates that contain neither a relevant primitive nor a relevant operation are dropped.

```

1 from pygraphviz import AGraph
2
3 import pygraphviz_mixin
4 from cvc_simplify import Simplify
5 from magic_predicates import get_predicates
6 from uppaal_nta import uppaal_xml
7 from stutter_bisimulation import quotient_partition, compute_quotient, merge_bisimilar_states, expand,
   remove_divergence
8 from operations import operations_to_edges, operations_to_separate_nodes
9
10 def magic_run(file, relevant_primitives, relevant_operations, variables, cvc_variables):
11     graph = AGraph(file + '.dot')
12     relevant = relevant_primitives ^ relevant_operations
13     operators = frozenset(['=', '==', '!=', '>', '>=', '<', '<=', '||', '+', '-', '[', ']'])
14
15     cvc_simplify = Simplify(variables, cvc_variables, operators)
16
17     # (1) Remove all graphs that are not an LTS
18     for subgraph in graph.subgraphs():
19         if not subgraph.name.startswith('cluster_pred_abs_lts_'):
20             for node in subgraph.nodes():
21                 graph.delete_node(node)
22             subgraph.clear()
23
24     # (2) Unlabel graph
25     for edge in graph.edges():
26         del edge.attr['label']
27
28     # (3,4) Remove irrelevant predicates and simplify relevant ones
29     def is_relevant_predicate(predicate):
30         return any(map(lambda primitive: primitive in relevant, predicate.split()))
31
32     for node in graph.nodes():
33         relevant_predicates = cvc_simplify.simplify_predicates(filter(is_relevant_predicate,
34                               get_predicates(node)))
35         node.set_label(relevant_predicates)
36
37     # (5) Move operations to their own nodes
38     operations_to_separate_nodes(graph, relevant_operations)
39
40     # (6) Quotienting
41     expand(graph) # To make quotient divergence sensitive
42     # Compute initial partition (as described in PMC, algorithm 29, page 472)
43     partition = quotient_partition(graph)
44     quotient = compute_quotient(graph, partition)
45     merge_bisimilar_states(graph, quotient)
46     remove_divergence(graph) # To make quotient divergence sensitive
47
48     # (7) Move operations to edges
49     operations_to_edges(graph, relevant_operations)
50
51     # (8) Generate XML for Uppaal
52     uppaal_xml(graph, variables, '%s-nta.xml' % file)

```

Listing 26: The script magic_run.py

3. Remove implied predicates: when CVC can prove that a predicate is implied by other predicates, this predicate is dropped. (For example, when both “ $x > 2$ ” and “ $x > 1$ ” are present, the latter one is dropped.)
4. Eliminate disjunction: when “ $A \parallel B$ ” holds and CVC can prove that “NOT A ” holds, then “ $A \parallel B$ ” is replaced with “ B ”.

E.4 DISCUSSION

We calculate the divergence sensitive stutter bisimulation, which maintains “CTL* minus next” properties. The “next” operator is only useful when the step size is well defined, which is not the case for code. Therefore, it is clear that we do not need to preserve the “next” operator during simplification. However, preserving all other properties is conservative. When fewer properties are preserved, the simplification can provide a smaller model. Therefore, it is future research to identify which properties are useful in practice to allow more simplification than MAGIC RUM performs.

F | PROFILED CEGAR

We have tested our CEGAR with profiling on the code shown in listing 27. Our tool chain creates UPPAAL models from Java bytecode through a series of transformations. Most of our transformation are written in the Epsilon Transformation Language (ETL) provided by the Eclipse plugin Epsilon¹. To create a model, we perform the following steps:

1. Run our JBCPP tool (developed by Christoph Bockisch) to create a JBCPP model.
2. Run the Epsilon transformation `groupSequences` to simplify the model by grouping sequences of bytecodes, as explained in section 9.2.1.
3. Run the Epsilon transformation `deleteUnconnectedNodes` (developed by Buğra M. Yildiz) to delete nodes that are part of the exceptional flow or that are inside groups, as these nodes cannot be handled properly by the `jbcpp2uppaal` transformation.
4. Run the Epsilon transformation `jbcpp2uppaal` (developed by Buğra M. Yildiz) to transform the JBCPP model to a UPPAAL model. The transformation will ask several questions: we perform no recursion removal, no loop handling and request three instances of both templates.
5. Optionally, run the Epsilon transformation `layoutUpaal` to copy the layout of another UPPAAL model to the newly generated one. Because the default layout is chaotic, the layout of the UPPAAL model is often improved by hand. This transformation can copy the layout of a previous iteration to the current one to avoid manual work on the layout of the model.
6. Run the Epsilon transformation `uppaal2xsd` (developed by Arend Rensink) to transform the UPPAAL model to an XML file that can be read by the UPPAAL tool.
7. Manually add parameters to the UPPAAL templates. The transformations cannot handle template parameters properly and such parameters are needed to create multiple instances of each template. For example, the template `test$DownloadActivity` should have the following parameter: `const pid_test$DownloadActivity pid`.
8. Verify desired properties with UPPAAL. Now we have a full UPPAAL model, on which we can check properties, such as the following ones:
 - There are no deadlocks: `A[] not deadlock`.
 - The system can finish within a certain time: `E<> test$DownloadActivity#main.finish && test$DownloadActivity#main.wallClock < 200`.

¹ Epsilon is available at eclipse.org/epsilon/.


```

1 import ...;
2
3 public class DownloadActivity {
4
5     private static final long PERIOD = 30 * 1000;
6     private Thread downloader;
7     private boolean download;
8
9     public void startDownload() {
10         downloader = new Thread("Downloader") {
11             @Override
12             public void run() {
13                 System.out.println("Outerloop starts");
14                 while (download) {
15                     try {
16                         long startTime = System.currentTimeMillis();
17                         download();
18                         long delta = System.currentTimeMillis() - startTime;
19                         if (delta < PERIOD) {
20                             sleep(PERIOD - delta);
21                         }
22                     } catch (IOException e) {
23                         e.printStackTrace();
24                     } catch (InterruptedException e) {
25                         e.printStackTrace();
26                     }
27                 }
28                 System.out.println("Outerloop finishes");
29             }
30         };
31
32         public void download() throws IOException {
33             URL url = new URL("http://ftp.nl.debian.org/debian/ls-lR.gz");
34             HttpURLConnection connection = (HttpURLConnection) url.openConnection();
35             connection.connect();
36
37             int responseCode = connection.getResponseCode();
38             if (responseCode != HttpURLConnection.HTTP_OK) {
39                 String message = connection.getResponseMessage();
40                 System.err.println("Server returned HTTP " + responseCode + " " + message);
41             } else {
42                 InputStream input = new BufferedInputStream(connection.getInputStream());
43                 System.out.println("Innerloop starts");
44                 while(input.read() != -1); // Read all content and discard it
45                 System.out.println("Innerloop finishes");
46             }
47             connection.disconnect();
48         }
49         download = true;
50         downloader.start();
51     }
52 }

```

Listing 27: Example Java class on which we apply CEGAR

As explained in section 9.2.3, we start with the outermost loop. Therefore, it might also be useful to provide key properties for this main loop. A possibility is to add an additional clock `mainLoopClock` and reset this clock when the main loop is entered. We can now specify a maximum time for an iteration of the main loop. For example, that unless the system is finished, the main loop is entered each 800 time units: `A[] test$DownloadActivity#main.finish || mainLoopClock < 800`.

After checking the key properties, the system should be profiled. Profiling is still largely manual, and can be achieved through the following steps:

9. Identify (outermost) loop in counterexample.
10. Add log statements on its enter and exit edges, such that the profiler can measure when the execution is inside the loop.
11. Run the profiler to acquire an energy profile together with the state of the system (as shown in figure 46 on page 116).
12. Analyze the output of the profiler:
 - This should provide bounds for the loop, such that the loop can be grouped into a group that specifies these bounds.
 - If no desired results are acquired, retry with the loops inside the profiled one, as explained in section 9.2.3.

With the results of profiling, the model can be refined. When bounds are acquired over a sequence of statements, these statements should be grouped, and the bounds should be specified of the group. This modifies the JBCPP model, after which we continue at step 3 to iteratively perform all steps again, until at step 8 all key properties are satisfied.

- [P.1] Steven te Brinke. “First-order function dispatch in a Java-like programming language”. MA thesis. Sup. by Christoph M. Bockisch, Lodewijk M. J. Bergmans, and Mehmet Akşit. Enschede, The Netherlands: University of Twente, Jan. 2011. URL: <http://essay.utwente.nl/60118/> (cit. on pp. 7, 17, 19, 31, 121).
- [P.2] Steven te Brinke, Lodewijk M. J. Bergmans, and Christoph M. Bockisch. “The Keyword Revolution: Promoting language constructs for data access to first class citizens”. In: *Proceedings of the 1st International Workshop on Free Composition (FREECO@ECOOP 2011)*. Lancaster, United Kingdom: ACM, July 2011, 4:1–4:5. ISBN: 978-1-4503-0892-2. DOI: [10.1145/2068776.2068780](https://doi.org/10.1145/2068776.2068780) (cit. on p. 31).
- [P.3] Steven te Brinke, Christoph M. Bockisch, and Lodewijk M. J. Bergmans. “Reuse of Continuation-based Control-flow Abstractions”. In: *Proceedings of the 2nd workshop on Free composition @ onward! (FREECO@Onward! 2011)*. Portland, Oregon, USA: ACM, Oct. 2011, pp. 13–18. ISBN: 978-1-4503-1025-3. DOI: [10.1145/2089172.2089176](https://doi.org/10.1145/2089172.2089176) (cit. on p. 45).
- [P.4] Somayeh Malakuti, Steven te Brinke, Lodewijk M. J. Bergmans, and Christoph M. Bockisch. “Towards Modular Resource-Aware Applications”. In: *Proceedings of the 3rd international workshop on Variability & Composition (VariComp 2012)*. Potsdam, Germany: ACM, Mar. 2012, pp. 13–17. ISBN: 978-1-4503-1101-4. DOI: [10.1145/2161996.2162001](https://doi.org/10.1145/2161996.2162001) (cit. on p. 63).
- [P.5] Steven te Brinke, Mark Laarakkers, Christoph M. Bockisch, and Lodewijk M. J. Bergmans. “An Implementation Mechanism for Tailorable Exceptional Flow”. In: *Proceedings of the 5th International Workshop on Exception Handling (WEH 2012)*. Zurich, Switzerland: IEEE Computer Society, June 2012, pp. 22–26. ISBN: 978-1-4673-1767-2. DOI: [10.1109/WEH.2012.6226594](https://doi.org/10.1109/WEH.2012.6226594) (cit. on p. 45).
- [P.6] Lodewijk M. J. Bergmans, Steven te Brinke, Christoph M. Bockisch, and Mehmet Akşit. “Free Composition Instead of Language Dictatorship”. In: *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*. Rome, Italy: SciTePress, July 2012, pp. 388–393. ISBN: 978-989-8565-19-8 (cit. on p. 23).
- [P.7] Andre Loker, Steven te Brinke, and Christoph M. Bockisch. “A Unified Model of Aspect-Instantiation Policies”. In: *Proceedings of the 3rd International Workshop on Free Composition (FREECO@SPLASH 2012)*. Tucson, AZ, USA: ACM, Oct. 2012, pp. 1–4. ISBN: 978-1-4503-1635-4. DOI: [10.1145/2414716.2414718](https://doi.org/10.1145/2414716.2414718) (cit. on p. 42).

- [P.8] Steven te Brinke, Somayeh Malakuti, Christoph M. Bockisch, Lodewijk M. J. Bergmans, and Mehmet Akşit. *A Design Method For Modular Energy-Aware Software*. Technical Report TR-CTIT-12-28. Enschede, The Netherlands: Centre for Telematics and Information Technology, University of Twente, Nov. 2012. URL: <http://purl.utwente.nl/publications/84364> (cit. on p. 77).
- [P.9] Steven te Brinke, Somayeh Malakuti, Christoph M. Bockisch, Lodewijk M. J. Bergmans, and Mehmet Akşit. “A Design Method For Modular Energy-Aware Software”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*. Coimbra, Portugal: ACM, Mar. 2013, pp. 1180–1182. ISBN: 978-1-4503-1656-9. DOI: [10.1145/2480362.2480584](https://doi.org/10.1145/2480362.2480584) (cit. on p. 77).
- [P.10] Steven te Brinke, Christoph M. Bockisch, Lodewijk M. J. Bergmans, Somayeh Malakuti, Mehmet Akşit, and Shmuel Katz. “Deriving Minimal Models for Resource Utilization”. In: *Proceedings of the 1st International Workshop Green In Software Engineering, Green By Software Engineering (GIBSE 2013)*. Fukuoka, Japan: ACM, Mar. 2013, pp. 15–18. ISBN: 978-1-4503-1866-2. DOI: [10.1145/2451605.2451612](https://doi.org/10.1145/2451605.2451612) (cit. on p. 93).
- [P.11] Steven te Brinke, Somayeh Malakuti, Christoph M. Bockisch, Lodewijk M. J. Bergmans, Mehmet Akşit, and Shmuel Katz. “A Tool-Supported Approach for Modular Design of Energy-Aware Software”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*. Gyeongju, Korea: ACM, Mar. 2014, pp. 1206–1212. ISBN: 978-1-4503-2469-4. DOI: [10.1145/2554850.2554964](https://doi.org/10.1145/2554850.2554964) (cit. on pp. 63, 93).
- [P.12] Steven te Brinke. “Interpreting Energy Profiles with CEGAR”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*. Gyeongju, Korea: ACM, Mar. 2014, pp. 1221–1222. ISBN: 978-1-4503-2469-4. DOI: [10.1145/2554850.2555194](https://doi.org/10.1145/2554850.2555194) (cit. on p. 105).
- [P.13] Brend Wanders and Steven te Brinke. “Strata: Typed Semi-Structured Data in DokuWiki”. In: *Proceedings of the 10th International Symposium on Open Collaboration (OpenSym 2014)*. Berlin, Germany: ACM, Aug. 2014, p. 50. ISBN: 978-1-4503-3016-9. DOI: [10.1145/2641580.2641636](https://doi.org/10.1145/2641580.2641636).
- [P.14] Jaco C. van de Pol, Theo C. Ruys, and Steven te Brinke. “Thoughtful Brute Force Attack of the RERS 2012 and 2013 Challenges”. In: *International journal on software tools for technology transfer* 16.5 (Oct. 2014), pp. 481–491. ISSN: 1433-2779. DOI: [10.1007/s10009-014-0324-3](https://doi.org/10.1007/s10009-014-0324-3).
- [P.15] Lesley Wevers and Steven te Brinke. “Monte-Carlo Tree Search for Poly-Y”. In: *Proceedings of the 26th Benelux Conference on Artificial Intelligence (BNAIC 2014)*. Nijmegen, the Netherlands: Radboud Universiteit Nijmegen, Nov. 2014, pp. 119–127.

- [T.1] Teun van Hemert. “Implementing flexible, extensible composition operators”. MA thesis. Sup. by Christoph M. Bockisch, Steven te Brinke, and Lodewijk M. J. Bergmans. Enschede, The Netherlands: University of Twente, June 2013. URL: <http://essay.utwente.nl/63405/> (cit. on p. 147).
- [T.2] Mark Laarakkers. “Generating Continuation Passing Style Code for the Co-op Language”. MA thesis. Sup. by Christoph M. Bockisch, Lodewijk M. J. Bergmans, and Steven te Brinke. Enschede, The Netherlands: University of Twente, June 2013. URL: <http://essay.utwente.nl/63432/> (cit. on p. 147).
- [T.3] Roel ter Maat. “A Framework for Modular Implementation of Domain-Specific Event-Based Applications”. MA thesis. Sup. by Somayeh Malakuti, Mehmet Akşit, Christoph M. Bockisch, and Steven te Brinke. Enschede, The Netherlands: University of Twente, Mar. 2014. URL: <http://essay.utwente.nl/64724/>.
- [T.4] Alexander Bakker. “Comparing Energy Profilers for Android”. In: *21st Twente Student Conference on IT*. Vol. 21. TSConIT. Enschede, The Netherlands: University of Twente, June 23, 2014 (cit. on p. 111).
- [T.5] Michael P. R. Hannema. “A Continuation-based Framework for Control Structures in Scheme”. In: *21st Twente Student Conference on IT*. Vol. 21. TSConIT. Enschede, The Netherlands: University of Twente, June 23, 2014 (cit. on p. 56).
- [T.6] Nick ten Veen. “Using Function Profiling for Implicit Parallelism”. In: *21st Twente Student Conference on IT*. Vol. 21. TSConIT. Enschede, The Netherlands: University of Twente, June 23, 2014.
- [T.7] Dennis Windhouwer. “The Modularity of the Resource Utilization Model in Python”. MA thesis. Sup. by Mehmet Akşit, Pim van den Broek, Christoph M. Bockisch, and Steven te Brinke. Enschede, The Netherlands: University of Twente, Dec. 2014. URL: <http://essay.utwente.nl/66433/> (cit. on pp. 2, 27).

BIBLIOGRAPHY

- [1] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. “Closed-Form Upper Bounds in Static Cost Analysis”. In: *Journal of Automated Reasoning* 46.2 (2011), pp. 161–203. ISSN: 0168-7433. DOI: [10.1007/s10817-010-9174-1](https://doi.org/10.1007/s10817-010-9174-1) (cit. on p. [111](#)).
- [2] Hany H. Ammar, Vittorio Cortellessa, and Alaa Ibrahim. “Modeling resources in a UML-based simulative environment”. In: *ACS/IEEE International Conference on Computer Systems and Applications*. Beirut, Lebanon, June 2001, pp. 405–410. DOI: [10.1109/AICCSA.2001.934026](https://doi.org/10.1109/AICCSA.2001.934026) (cit. on p. [63](#)).
- [3] Andrew W. Appel and Trevor Jim. “Continuation-Passing, Closure-Passing Style”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA, 1989, pp. 293–302. ISBN: 0-89791-294-2. DOI: [10.1145/75277.75303](https://doi.org/10.1145/75277.75303) (cit. on p. [59](#)).
- [4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press Cambridge, 2008. ISBN: 978-0-262-02649-9 (cit. on p. [100](#)).
- [5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. “CVC4”. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. ISBN: 978-3-642-22109-5. DOI: [10.1007/978-3-642-22110-1_14](https://doi.org/10.1007/978-3-642-22110-1_14) (cit. on p. [100](#)).
- [6] Clark Barrett and Cesare Tinelli. “CVC3”. In: *Computer Aided Verification*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 298–302. ISBN: 978-3-540-73367-6. DOI: [10.1007/978-3-540-73368-3_34](https://doi.org/10.1007/978-3-540-73368-3_34) (cit. on p. [100](#)).
- [7] Gerd Behrmann, Alexandre David, and Kim Larsen. “A Tutorial on UPPAAL”. In: *Formal Methods for the Design of Real-Time Systems*. Ed. by Marco Bernardo and Flavio Corradini. Vol. 3185. Lecture Notes in Computer Science. Springer, 2004, pp. 33–35. ISBN: 978-3-540-23068-7. DOI: [10.1007/978-3-540-30080-9_7](https://doi.org/10.1007/978-3-540-30080-9_7) (cit. on pp. [77](#), [83](#)).
- [8] Luca Benini, Robin Hodgson, and Polly Siegel. “System-level Power Estimation and Optimization”. In: *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*. ISLPED ’98. Monterey, California, USA: ACM, Aug. 1998, pp. 173–178. ISBN: 1-58113-059-7. DOI: [10.1145/280756.280881](https://doi.org/10.1145/280756.280881) (cit. on p. [66](#)).
- [9] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep into Pharo*. 1st ed. Switzerland: Square Bracket Associates, Aug. 2013. Chap. 13. ISBN: 978-3-9523341-6-4. URL: <http://www.deepintopharo.com> (cit. on p. [59](#)).

- [10] Lodewijk M.J. Bergmans, Wilke K. Havinga, and Mehmet Akşit. “First-class Compositions: Defining and composing object and aspect compositions with first-class operators”. In: *Transactions on Aspect-Oriented Software Development IX*. Lecture Notes in Computer Science 7271 (2012). Ed. by Gary T. Leavens, Shigeru Chiba, Michael Haupt, Klaus Ostermann, and Eric Wohlstadter, pp. 216–267. DOI: [10.1007/978-3-642-35551-6_6](https://doi.org/10.1007/978-3-642-35551-6_6) (cit. on pp. 28, 30).
- [11] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. “The software model checker BLAST: Applications to software engineering”. In: *Int. J. Softw. Tools Technol. Transf.* 9.5 (Oct. 2007), pp. 505–525. ISSN: 1433-2779. DOI: [10.1007/s10009-007-0044-z](https://doi.org/10.1007/s10009-007-0044-z) (cit. on p. 95).
- [12] Christoph C. M. Bockisch, Andreas Sewe, Haihan Yin, Mira Mezini, and Mehmet Akşit. “An In-Depth Look at ALIA₄”. In: *Journal of Object Technology* 11.1 (Apr. 2012), 7:1–28. ISSN: 1660-1769. DOI: [10.5381/jot.2012.11.1.a7](https://doi.org/10.5381/jot.2012.11.1.a7) (cit. on p. 147).
- [13] Atze G. Bouius. “Characterising the Ripple Effects of Introducing Energy-Awareness Functionality in Cyber-Physical-Systems Software in Cyber-Physical System software”. MA thesis. Sup. by Mehmet Akşit, Pim van den Broek, Jan Broenink, Somayeh Malakuti, and Valter Vieira de Camargo. Enschede, The Netherlands: University of Twente, Jan. 2015. URL: <http://essay.utwente.nl/66659/> (cit. on p. 27).
- [14] Gilad Bracha and Gary Lindstrom. “Modularity Meets Inheritance”. In: *In Proc. International Conference on Computer Languages*. IEEE Computer Society, Apr. 1992, pp. 282–290. DOI: [10.1109/ICCL.1992.185492](https://doi.org/10.1109/ICCL.1992.185492) (cit. on p. 42).
- [15] Johan Brichau, Michael Haupt, Nicolas Leidenfrost, Awais Rashid, and Lodewijk et al. Bergmans. *Report describing survey of aspect languages and models*. Tech. rep. AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01. Vrije Universiteit Brussel, May 17, 2005 (cit. on p. 25).
- [16] Johan Brichau, Mira Mezini, Jacques Noyé, Wilke K. Havinga, Lodewijk M. J. Bergmans, Vaidas Gasiunas, Christoph M. Bockisch, Johan Fabry, and Theo D’Hondt. *An Initial Metamodel for Aspect-Oriented Languages*. Tech. rep. AOSD-Europe, Feb. 2006 (cit. on p. 59).
- [17] Frederick P. Brooks. “No silver bullet: Essence and accidents of software engineering”. In: *IEEE computer* (Apr. 1987) (cit. on p. 63).
- [18] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. “Modular verification of software components in C”. In: *Transactions on Software Engineering (TSE)* 30.6 (June 2004), pp. 388–402. ISSN: 0098-5589. DOI: [10.1109/TSE.2004.22](https://doi.org/10.1109/TSE.2004.22) (cit. on pp. 95, 99).
- [19] Sagar Chaki, Edmund M. Clarke, Alex Groce, Joel Ouaknine, Ofer Strichman, and Karen Yorav. “Efficient verification of sequential and concurrent C programs”. In: *Formal Methods in System Design (FMSD)* 25.2-3 (2004), pp. 129–166. ISSN: 0925-9856. DOI: [10.1023/B:FORM.0000040026.56959.91](https://doi.org/10.1023/B:FORM.0000040026.56959.91) (cit. on pp. 95, 99).

- [20] Sagar Chaki, Edmund M. Clarke, Alex Groce, and Ofer Strichman. “Predicate Abstraction with Minimum Predicates”. In: *Correct Hardware Des. and Verification Methods*. Ed. by Daniel Geist and Enrico Tronci. Vol. 2860. Lecture Notes in Computer Science. Springer, 2003, pp. 19–34. ISBN: 978-3-540-20363-6. DOI: [10.1007/978-3-540-39724-3_5](#) (cit. on p. [99](#)).
- [21] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. *Software Engineering for Self-Adaptive Systems*. Vol. 5525. Lecture Notes in Computer Science. Springer, 2009. ISBN: 978-3-642-02160-2 (cit. on p. [2](#)).
- [22] Alonzo Church and J. B. Rosser. “Some Properties of Conversion”. In: *Transactions of the American Mathematical Society* 39.3 (May 1936), pp. 472–482. ISSN: 0002-9947. URL: <http://www.jstor.org/stable/1989762> (cit. on p. [15](#)).
- [23] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”. In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263. ISSN: 0164-0925. DOI: [10.1145/5397.5399](#) (cit. on p. [72](#)).
- [24] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169. ISBN: 978-3-540-67770-3. DOI: [10.1007/10722167_15](#) (cit. on p. [93](#)).
- [25] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. “SAT-ABS: SAT-based Predicate Abstraction for ANSI-C”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 570–574. ISBN: 978-3-540-25333-4. DOI: [10.1007/978-3-540-31980-1_40](#) (cit. on pp. [95](#), [96](#)).
- [26] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. “MultiJava: modular open classes and symmetric multiple dispatch for Java”. In: *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA ’00. Minneapolis, Minnesota, United States: ACM, 2000, pp. 130–145. ISBN: 1-58113-200-X. DOI: [10.1145/353171.353181](#) (cit. on p. [42](#)).
- [27] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. “Energy types”. In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 831–850. ISSN: 0362-1340. DOI: [10.1145/2398857.2384676](#) (cit. on p. [101](#)).
- [28] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare, eds. *Structured Programming*. London, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3 (cit. on p. [59](#)).
- [29] Olivier Danvy. “On Evaluation Contexts, Continuations, and the Rest of the Computation”. In: *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW’04)*. Ed. by Hayo Thielecke. CSR-04-1. Birmingham B15 2TT, United Kingdom, Jan. 2004 (cit. on p. [48](#)).

- [30] Alexandre David, M. Oliver Möller, and Wang Yi. “Formal Verification of UML Statecharts with Real-Time Extensions”. In: *Fundamental Approaches to Software Engineering*. Ed. by Ralf-Detlef Kutsche and Herbert Weber. Vol. 2306. Lecture Notes in Computer Science. Springer, 2002, pp. 208–241. ISBN: 978-3-540-43353-8. DOI: [10.1007/3-540-45923-5_15](#) (cit. on p. [91](#)).
- [31] Edsger W. Dijkstra. *A discipline of programming*. Englewood Cliffs, New Jersey: Prentice-Hall, 1976 (cit. on p. [65](#)).
- [32] Cynthia Disenfeld and Shmuel Katz. “Specification and verification of event detectors and responses”. In: *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development (AOSD)*. Fukuoka, Japan: ACM, 2013, pp. 121–132. ISBN: 978-1-4503-1766-5. DOI: [10.1145/2451436.2451452](#) (cit. on p. [95](#)).
- [33] Bo Einarsson and W. Morven Gentleman. “Mixed Language Programming”. In: *Software: Practice and Experience* 14.4 (Apr. 1984), pp. 383–395. ISSN: 1097-024X. DOI: [10.1002/spe.4380140410](#) (cit. on p. [43](#)).
- [34] Carla Schlatter Ellis. “The Case for Higher-Level Power Management”. In: *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems. HOTOS '99*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 162–167. ISBN: 0-7695-0237-7. DOI: [10.1109/HOTOS.1999.798394](#) (cit. on pp. [1](#), [67](#)).
- [35] Yusuke Endoh, Hidehiko Masuhara, and Akinori Yonezawa. “Continuation join points”. In: *Workshop on Foundations of Aspect Oriented Languages*. Bonn, Germany, Mar. 2006, pp. 1–10 (cit. on p. [55](#)).
- [36] Michael Ernst, Craig Kaplan, and Craig Chambers. “Predicate dispatching: A unified theory of dispatch”. In: *Lecture Notes in Computer Science* 1445 (1998). Ed. by Eric Jul, pp. 186–211. DOI: [10.1007/BFb0054092](#) (cit. on p. [60](#)).
- [37] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer M. Anderson. “Quantifying the energy consumption of a pocket computer and a Java virtual machine”. In: *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. Vol. 28. SIGMETRICS '00 1. New York, NY, USA: ACM, June 2000, pp. 252–263. ISBN: 1-58113-194-1. DOI: [10.1145/339331.339421](#) (cit. on p. [1](#)).
- [38] Yungsi Fei, Lin Zhong, and Niraj K. Jha. “An energy-aware framework for dynamic software management in mobile computing systems”. In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 27:1–27:31. ISSN: 1539-9087. DOI: [10.1145/1347375.1347380](#) (cit. on p. [102](#)).
- [39] Miguel A. Ferreira, Eric Hoekstra, Bo Merkus, Bram Visser, and Joost Visser. “SE-FLab: A Lab for Measuring Software Energy Footprints”. In: *2nd International Workshop on Green and Sustainable Software (GREENS)*. May 2013, pp. 30–37. DOI: [10.1109/GREENS.2013.6606419](#) (cit. on p. [111](#)).
- [40] Andrzej Filinski. “Representing monads”. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '94. Portland, Oregon, United States: ACM, 1994, pp. 446–457. ISBN: 0-89791-636-0. DOI: [10.1145/174675.178047](#) (cit. on p. [55](#)).

- [41] Jason Flinn and M. Satyanarayanan. “Energy-aware adaptation for mobile applications”. In: *Proc. 17th ACM Sympos. on Operating Syst. Principles*. SOSP ’99. Charleston, South Carolina, USA: ACM, 1999, pp. 48–63. ISBN: 1-58113-140-2. DOI: [10.1145/319151.319155](#) (cit. on p. [102](#)).
- [42] Jason Flinn and M. Satyanarayanan. “PowerScope: a tool for profiling the energy usage of mobile applications”. In: *Proc. 2nd IEEE Workshop on Mobile Comput. Syst. and Appl.* WMCSA ’99. Feb. 1999, pp. 2–10. DOI: [10.1109/MCSA.1999.749272](#) (cit. on p. [102](#)).
- [43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley, 1994. ISBN: 0-20130-953-X (cit. on p. [24](#)).
- [44] Jeff Garland and Richard Anthony. *Large-Scale Software Architecture: A Practical Guide using UML*. 1st. Wiley, 2003. ISBN: 0-470-84849-9 (cit. on p. [77](#)).
- [45] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6 (cit. on p. [28](#)).
- [46] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java™ Language Specification: Java SE 7 Edition*. Oracle, Feb. 2012 (cit. on p. [47](#)).
- [47] Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Assmann. “Architecture and Mechanisms for Energy Auto Tuning”. In: *Proc. Sustainable ICTs and Management Systems for Green Computing*. 2012, pp. 45–73. DOI: [10.4018/978-1-4666-1839-8.ch003](#) (cit. on pp. [77](#), [101](#), [102](#)).
- [48] Marcus Hähnel, Björn Döbel, Marcus Völz, and Hermann Härtig. “Measuring energy consumption for short code paths using RAPL”. In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (Jan. 2012), pp. 13–17. ISSN: 0163-5999. DOI: [10.1145/2425248.2425252](#) (cit. on p. [102](#)).
- [49] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. “Estimating mobile application energy consumption using program analysis”. In: *Proceedings of the 35th International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, May 2013, pp. 92–101. ISBN: 978-1-4673-3076-3. DOI: [10.1109/ICSE.2013.6606555](#) (cit. on p. [102](#)).
- [50] Wilke K. Havinga. “On the Design of Software Composition Mechanisms and the Analysis of Composition Conflicts”. PhD thesis. Enschede, The Netherlands: Twente University, June 2009. ISBN: 978-90-365-2842-9. DOI: [10.3990/1.9789036528429](#) (cit. on pp. [27](#), [40](#)).
- [51] Wilke K. Havinga, Lodewijk M. J. Bergmans, and Mehmet Akşit. “A Model for Composable Composition Operators: Expressing object and aspect compositions with first-class operators”. In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. AOSD ’10. Rennes and Saint-Malo, France: ACM, Mar. 2010, pp. 145–156. ISBN: 978-1-60558-958-9. DOI: [10.1145/1739230.1739248](#) (cit. on pp. [27](#), [30](#)).

- [52] Wilke K. Havinga, Christoph M. Bockisch, and Lodewijk M. J. Bergmans. “A Case for Custom, Composable Composition Operators”. In: *Proceedings of International Workshop on Composition & Variability*. Workshop Proceedings. Rennes, France: CEUR-WS, 2010, pp. 45–50 (cit. on pp. 30, 40).
- [53] Hagen Höpfner, Maximilian Schirmer, and Christian Bunse. “On Measuring Smartphones’ Software Energy Requirements”. In: *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*. Rome, Italy: SciTePress, July 2012, pp. 165–171 (cit. on p. 112).
- [54] Paul Hudak. “Building Domain-specific Embedded Languages”. In: *ACM Comput. Surv.* 28.4es (Dec. 1996). ISSN: 0360-0300. DOI: 10.1145/242224.242477 (cit. on p. 27).
- [55] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. “Report on the Programming Language Haskell: A Non-strict, Purely Functional Language”. In: *SIGPLAN Not.* 27 (5 May 1992), pp. 1–164. ISSN: 0362-1340. DOI: 10.1145/130697.130699 (cit. on p. 121).
- [56] Samuel N. Kamin. “Research on Domain-specific Embedded Languages and Program Generators”. In: *Electronic Notes in Theoretical Computer Science* 14 (1998). US-Brazil Joint Workshops on the Formal Foundations of Software Systems, pp. 149–168. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)80235-X (cit. on p. 27).
- [57] Gregor Kiczales. “It’s Not Metaprogramming”. In: *Software Development Magazine* 10 (Oct. 2004). URL: <http://www.drdobbs.com/its-not-metaprogramming/184415220> (cit. on p. 26).
- [58] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming”. In: *ECOOP’97 – Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoaka. Vol. 1241. Lecture Notes in Computer Science. Springer, 1997, pp. 220–242. ISBN: 978-3-540-63089-0. DOI: 10.1007/BFb0053381 (cit. on p. 24).
- [59] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: MIT Press, 1991. ISBN: 0-262-11158-6 (cit. on p. 26).
- [60] Stephen Cole Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. Wolters-Noordhoff, 1971. ISBN: 978-0-444-10088-7 (cit. on p. 15).
- [61] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. “ORBIT: An Optimizing Compiler for Scheme”. In: *SIGPLAN Not.* 21.7 (July 1986), pp. 219–233. ISSN: 0362-1340. DOI: 10.1145/13310.13333 (cit. on p. 59).
- [62] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented Programming in the BETA Programming Language*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. ISBN: 0-201-62430-3 (cit. on p. 28).

- [63] Umer Liqat, Steve Kerrison, Alejandro Serrano, Kyriakos Georgiou, Pedro Lopez-Garcia, Neville Grech, Manuel V. Hermenegildo, and Kerstin Eder. “Energy Consumption Analysis of Programs Based on XMOs ISA-Level Models”. In: *Logic-Based Program Synthesis and Transformation*. Ed. by Gopal Gupta and Ricardo Peña-Marí. Vol. 8901. Lecture Notes in Computer Science. Springer, 2014, pp. 72–90. ISBN: 978-3-319-14124-4. DOI: [10.1007/978-3-319-14125-1_5](https://doi.org/10.1007/978-3-319-14125-1_5) (cit. on p. [111](#)).
- [64] Somayeh Malakuti and Mehmet Akşit. “Event-Based Modularization of Reactive Systems”. In: *Concurrent Objects and Beyond*. Ed. by Gul Agha, Atsushi Igarashi, Naoki Kobayashi, Hidehiko Masuhara, Satoshi Matsuoka, Etsuya Shibayama, and Kenjiro Taura. Vol. 8665. Lecture Notes in Computer Science. Springer, 2014, pp. 367–407. ISBN: 978-3-662-44470-2. DOI: [10.1007/978-3-662-44471-9_16](https://doi.org/10.1007/978-3-662-44471-9_16) (cit. on p. [43](#)).
- [65] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York, NY, USA: Springer, 1992. ISBN: 978-1-4612-0931-7. DOI: [10.1007/978-1-4612-0931-7](https://doi.org/10.1007/978-1-4612-0931-7) (cit. on pp. [72](#), [97](#)).
- [66] Douwe van der Meij. *AOP in Python API design*. PyGrunn Symposium. Groningen, Netherlands, May 11, 2012. URL: <http://www.slideshare.net/meij200/aop-in-python-api-design-12937607> (cit. on p. [27](#)).
- [67] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and How to Develop Domain-specific Languages”. In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: [10.1145/1118890.1118892](https://doi.org/10.1145/1118890.1118892) (cit. on p. [27](#)).
- [68] Bertrand Meyer. *Eiffel: the language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN: 0-13-247925-7 (cit. on p. [53](#)).
- [69] István Nagy. “On the Design of Aspect-Oriented Composition Models for Software Evolution”. PhD thesis. Enschede, The Netherlands: University of Twente, June 2006. ISBN: 90-365-2368-0. URL: <http://doc.utwente.nl/60766/> (cit. on p. [18](#)).
- [70] Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. “A Theory of Name Resolution”. In: *Programming Languages and Systems*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 205–231. ISBN: 978-3-662-46668-1. DOI: [10.1007/978-3-662-46669-8_9](https://doi.org/10.1007/978-3-662-46669-8_9) (cit. on p. [60](#)).
- [71] Martin Odersky. *The Scala Language Specification, Version 2.9*. Programming Methods Laboratory, May 2011 (cit. on pp. [48](#), [147](#)).
- [72] OMG. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Tech. rep. formal/2009-11-02. OMG, 2011 (cit. on p. [63](#)).
- [73] Klaus Ostermann and Mira Mezini. “Object-oriented composition untangled”. In: *Proc. OOPSLA ’01 Conf. Object Oriented Programming Systems Languages and Applications*. Tampa Bay, FL, USA: ACM Press, 2001, pp. 283–299. DOI: [10.1145/504282.504303](https://doi.org/10.1145/504282.504303) (cit. on pp. [24](#), [41](#)).
- [74] David L. Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: [10.1145/361598.361623](https://doi.org/10.1145/361598.361623) (cit. on pp. [1](#), [77](#)).

- [75] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. “Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof”. In: *Proc. 7th ACM Eur. Conf. Comput. Syst. EuroSys ’12*. Bern, Switzerland: ACM, 2012, pp. 29–42. ISBN: 978-1-4503-1223-3. DOI: [10.1145/2168836.2168841](#) (cit. on p. [102](#)).
- [76] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. “Fine-grained power modeling for smartphones using system call tracing”. In: *Proc. 6th Conf. Comput. Syst. EuroSys ’11*. Salzburg, Austria: ACM, 2011, pp. 153–168. ISBN: 978-1-4503-0634-8. DOI: [10.1145/1966445.1966460](#) (cit. on p. [102](#)).
- [77] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. “A Multi-model Framework to Implement Self-managing Control Systems for QoS Management”. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS ’11*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 218–227. DOI: [10.1145/1988008.1988040](#) (cit. on p. [63](#)).
- [78] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. “The Glasgow Haskell compiler: a technical overview”. In: *UK Joint Framework for Information Technology (JFIT)*. Keele, Mar. 1993 (cit. on p. [121](#)).
- [79] Ian Piumarta. “An association-based model of dynamic behaviour”. In: *Proceedings of the 1st International Workshop on Free Composition. FREECO ’11*. Lancaster, UK: ACM, July 2011, 3:1–3:5. ISBN: 978-1-4503-0892-2. DOI: [10.1145/2068776.2068779](#) (cit. on p. [60](#)).
- [80] Christopher Poskitt, Mike Dodds, Richard F. Paige, and Arend Rensink. “Towards Rigorously Faking Bidirectional Model Transformations”. In: *Proceedings of the Workshop on Analysis of Model Transformations, AMT 2014, Valencia, Spain*. Ed. by J. Dingel, J. De Lara, L. Lúcio, and H. Vangheluwe. Vol. 1277. CEUR-WS. Valencia, Spain: RWTH Aachen, Germany, Sept. 2014, pp. 70–75 (cit. on p. [27](#)).
- [81] Feng Qian, Zhaoguang Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. “TOP: Tail Optimization Protocol For Cellular Radio Resource Allocation”. In: *18th IEEE Int. Conf. Netw. Protocols (ICNP)*. Oct. 2010, pp. 285–294. DOI: [10.1109/ICNP.2010.5762777](#) (cit. on pp. [1](#), [69](#), [149](#)).
- [82] Roshan Ramachandran, David J. Pearce, and Ian Welch. “AspectJ for Multilevel Security”. In: *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. Ed. by Yvonne Coady, David H. Lorenz, Olaf Spinczyk, and Eric Wohlstadter. Bonn, Germany: Published as University of Virginia Computer Science Technical Report CS–2006–01, Mar. 2006, pp. 13–17 (cit. on p. [42](#)).
- [83] Tiark Rompf, Ingo Maier, and Martin Odersky. “Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform”. In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming. ICFP ’09*. New York, NY, USA: ACM, 2009, pp. 317–328. ISBN: 978-1-60558-332-7. DOI: [10.1145/1596550.1596596](#) (cit. on pp. [49](#), [54](#), [55](#)).

- [84] Arjan J. de Roo, Michiel F. H. Hendriks, Wilke K. Havinga, Pascal E. A. Durr, and Lodewijk M. J. Bergmans. “Compose*: a Language- and Platform-Independent Aspect Compiler for Composition Filters”. In: *First International Workshop on Academic Software Development Tools and Techniques*. Ed. by Kim Mens, Mark van den Brand, Adrian Kuhn, Holger M. Kienle, and Roel Wuyts. WASDeTT ’08. Paphos, Cyprus, July 2008 (cit. on p. 42).
- [85] Guido van Rossum. *The Python Language Reference: Release 3.4.2*. Ed. by Fred L. Drake. Python Software Foundation, Oct. 9, 2014 (cit. on p. 27).
- [86] Walker Royce. *Improving Software Economics-Top 10 Principles of Achieving Agility at Scale*. White paper. IBM Rational, May 2009 (cit. on pp. 1, 63).
- [87] Cagri Sahin, Furkan Cayci, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. “Towards power reduction through improved software design”. In: *Energytech, 2012 IEEE*. May 2012, pp. 1–6. DOI: [10.1109/EnergyTech.2012.6304705](#) (cit. on p. 101).
- [88] Curt Schurgers, Vijay Raghunathan, and Mani B. Srivastava. “Modulation scaling for real-time energy aware packet scheduling”. In: *Global Telecommunications Conference*. Vol. 6. GLOBECOM ’01. IEEE, 2001, pp. 3653–3657. DOI: [10.1109/GLOCOM.2001.966363](#) (cit. on p. 1).
- [89] Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. “REMES: A Resource Model for Embedded Systems”. In: *14th IEEE Int. Conf. Eng. Complex Comput. Syst.* June 2009, pp. 84–94. DOI: [10.1109/ICECCS.2009.49](#) (cit. on p. 63).
- [90] Mary Shaw. “Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status”. In: *Studies of Software Design*. Ed. by David Alex Lamb. Vol. 1078. Lecture Notes in Computer Science. Springer, 1996, pp. 17–32. ISBN: 978-3-540-61285-8. DOI: [10.1007/BFb0030517](#) (cit. on pp. 23, 24).
- [91] Bob Steigerwald and Abhishek Agrawal. *Developing Green Software*. Tech. rep. Folsom, CA, USA: Intel Corporation, 2011, pp. 1–11 (cit. on p. 1).
- [92] Perdita Stevens. “Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions”. In: *Model Driven Engineering Languages and Systems*. Ed. by Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil. Vol. 4735. Lecture Notes in Computer Science. Springer, 2007, pp. 1–15. ISBN: 978-3-540-75208-0. DOI: [10.1007/978-3-540-75209-7_1](#) (cit. on p. 27).
- [93] Christopher Strachey and Christopher P. Wadsworth. “Continuations: A Mathematical Semantics for Handling Full Jumps”. In: *Higher-Order and Symbolic Computation* 13.1-2 (Apr. 2000), pp. 135–152. ISSN: 1388-3690. DOI: [10.1023/A:1010026413531](#) (cit. on p. 47).
- [94] Clemens Szyperski. *Component software: beyond object-oriented programming*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1998. ISBN: 0-201-17888-5 (cit. on p. 64).
- [95] Antero Taivalsaari. “On the notion of inheritance”. In: *ACM Comput. Surv.* 28.3 (1996), pp. 438–479. ISSN: 0360-0300. DOI: [10.1145/243439.243441](#) (cit. on pp. 25, 36, 39).

- [96] Éric Tanter. “Aspects of Composition in the Reflex AOP Kernel”. In: *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*. Vol. 4089. Lecture Notes in Computer Science. Vienna, Austria: Springer, Mar. 2006, pp. 98–113. DOI: [10.1007/11821946_7](https://doi.org/10.1007/11821946_7) (cit. on p. 42).
- [97] Éric Tanter and Johan Fabry. “Supporting Composition of Structural Aspects in an AOP Kernel”. In: *Journal of Universal Computer Science* 15.3 (Feb. 2009), pp. 620–647. DOI: [10.3217/jucs-015-03-0620](https://doi.org/10.3217/jucs-015-03-0620) (cit. on p. 42).
- [98] Narayanan Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin, Hyun Suk Kim, and Wu Ye. “Energy-driven Integrated Hardware-software Optimizations Using SimplePower”. In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ISCA ’00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 95–106. ISBN: 1-58113-232-8. DOI: [10.1145/339647.339659](https://doi.org/10.1145/339647.339659) (cit. on p. 101).
- [99] Dennis Wagelaar and Lodewijk M. J. Bergmans. “Using a Concept-Based Approach to Aspect-Oriented Software Design”. In: *Workshop on Aspect Oriented Design — Identifying, Separating and Verifying Concerns in the Design (AOSD-2002)*. Ed. by Maurice Glandrup, Peri Tarr, Siobhán Clarke, and Faisal Akkawi. Enschede, The Netherlands, Mar. 2002 (cit. on p. 24).
- [100] Ken Wakita. “First class messages as first class continuations”. In: *Object Technologies for Advanced Software*. Ed. by Shojiro Nishio and Akinori Yonezawa. Vol. 742. Lecture Notes in Computer Science. Springer, 1993, pp. 442–459. ISBN: 978-3-540-57342-5. DOI: [10.1007/3-540-57342-9_88](https://doi.org/10.1007/3-540-57342-9_88) (cit. on p. 59).
- [101] Ching-lin Wang. “Obtaining Lazy Evaluation with Continuations in Scheme”. In: *Information Processing Letters* 35 (2 1990), pp. 93–97. DOI: [10.1016/0020-0190\(90\)90112-B](https://doi.org/10.1016/0020-0190(90)90112-B) (cit. on p. 55).
- [102] David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2006. ISBN: 978-0-470-02047-0 (cit. on p. 15).
- [103] Peter Wegner. “Concepts and Paradigms of Object-oriented Programming”. In: *SIG-PLAN OOPS Mess.* 1.1 (Aug. 1990), pp. 7–87. ISSN: 1055-6400. DOI: [10.1145/382192.383004](https://doi.org/10.1145/382192.383004) (cit. on p. 9).
- [104] Claas Wilke, Sebastian Götz, and Sebastian Richly. “JouleUnit: a generic framework for software energy profiling and testing”. In: *Proc. 2013 Workshop on Green in/by Softw. Eng. GIBSE ’13*. Fukuoka, Japan: ACM, Mar. 2013, pp. 9–14. ISBN: 978-1-4503-1866-2. DOI: [10.1145/2451605.2451610](https://doi.org/10.1145/2451605.2451610) (cit. on p. 111).
- [105] Wolfgang Wilkes. “Instance Inheritance Mechanisms for Object Oriented Databases”. In: *Advances in Object-Oriented Database Systems*. Ed. by Klaus R. Dittrich. Vol. 334. Lecture Notes in Computer Science. Springer, 1988, pp. 274–279. ISBN: 978-3-540-50345-3. DOI: [10.1007/3-540-50345-5_24](https://doi.org/10.1007/3-540-50345-5_24) (cit. on p. 39).
- [106] Xerox Corporation. *The AspectJ Programming Guide*. URL: <https://www.eclipse.org/aspectj/doc/released/progguide/> (cit. on p. 42).

- [107] Haihan Yin, Christoph M. Bockisch, and Mehmet Akşit. “A fine-grained, customizable debugger for aspect-oriented programming”. In: *Transactions on Aspect-Oriented Software Development X*. Lecture Notes in Computer Science 7800 (2013). Ed. by Gary T. Leavens, Shigeru Chiba, and Éric Tanter, pp. 1–38. ISSN: 1864-3027. DOI: [10.1007/978-3-642-36964-3_1](https://doi.org/10.1007/978-3-642-36964-3_1) (cit. on p. [26](#)).
- [108] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. “ECOSystem: managing energy as a first class operating system resource”. In: *SIGOPS Oper. Syst. Rev.* 36.5 (Oct. 2002), pp. 123–132. ISSN: 0163-5980. DOI: [10.1145/635508.605411](https://doi.org/10.1145/635508.605411) (cit. on p. [101](#)).

SUMMARY

Awareness of environmental sustainability, together with an increasing use of software, makes optimization of software energy consumption evermore important. Energy is one of many resources that is managed by software, and reducing energy consumption cannot be considered without taking into account the trade-offs with other resources and services.

Optimization techniques, implemented in software, can lead to substantial reduction of resource consumption, within both the computer system and the system being controlled. This thesis focusses on the reduction of energy consumption by controlling various external hardware components that are energy intensive.

Today's software faces the problem of complexity, and modularization is considered as means to cope with the complexity of software. Therefore, we modularized energy optimizers from the rest of the software. This thesis introduces a modeling notation that supports energy behavior. Our notation is used to understand the energy behavior of the system and to analyze which control strategy reduces the energy consumption. We also present guidelines for designing models that contain energy behavior and present a method to automatically extract models, including energy behavior, from source code together with profiling of the application. These Resource-Utilization Models (RUMs) are over-abstractions of the application and can be analyzed formally, for example, with the model checker UPPAAL.

Energy is actually consumed during the execution of the software. Therefore, it is inevitable to carry out optimizations at runtime. In order to support such software design, it must be possible to reuse the energy models at runtime without compromising the modularity of the implementation. However, fully supporting our models in a *mainstream* programming language reduces the readability of the source code. Therefore, this thesis introduces a programming language, named Co-op, that aims to address this issue. Co-op can be positioned between Aspect-Oriented Programming (AOP) and a Meta-Object Protocol (MOP): it is less expressive than a full MOP, but more expressive than AOP. This way, it supports all operations needed for implementing our models, because as well too expressive as too limiting languages reduce the readability of the source code.

Combining design and implementation support for Resource-Utilization Models allows software developers to optimize the energy consumption of their software in a modular way. This reduces the energy footprint of software while improving its maintainability and understandability. Hopefully, our approach will be adopted in practice to create a better planet with higher quality software.

SAMENVATTING

Bewustwording van duurzaamheid, samen met een toenemend gebruik van software, maakt optimalisatie van het energiegebruik van software steeds belangrijker. Energie is één van de vele hulpbronnen waarvan het gebruik door software wordt beïnvloed. Daarom omvat het verminderen van energiegebruik tevens compromissen ten aanzien van het gebruik van andere hulpbronnen en diensten.

Optimalisatietechnieken die in software zijn geïmplementeerd, kunnen een substantiële afname van het energiegebruik bewerkstelligen, zowel in het computersysteem als in het systeem dat wordt aangestuurd. Dit proefschrift focust op de energievermindering die kan worden bewerkstelligd door goede aansturing van externe hardwarecomponenten die veel energie gebruiken.

De software van vandaag kampt met complexiteitsproblemen. Modularisatie kan worden gebruikt om deze problemen te verminderen. Daarom scheiden wij optimalisaties van de rest van de software door deze modulair te specificeren. Dit proefschrift introduceert een modeleringsnotatie waarin tevens energiegebruik kan worden gespecificeerd. Onze notatie wordt gebruikt om het energiegedrag van het systeem te begrijpen, en te analyseren welke strategie gebruikt kan worden om het energiegebruik zo veel mogelijk te reduceren. We presenteren tevens richtlijnen voor het ontwerpen van onze modellen en presenteren een methode om zulke modellen, inclusief het energiegebruik, automatisch te extraheren uit broncode, aangevuld met energieprofielen die kunnen worden verkregen met software profilers. Onze modellen, genaamd Resource-Utilization Models (RUMs), zijn over-abstracties van de applicatie en kunnen formeel worden geanalyseerd, bijvoorbeeld met behulp van de model checker UPPAAL.

Energie wordt pas daadwerkelijk verbruikt tijdens het uitvoeren van software. Daarom is het onvermijdelijk om, naast het optimaliseren van het softwareontwerp, te optimaliseren tijdens het draaien van de software. Hiervoor moet het mogelijk zijn de RUMs te hergebruiken tijdens het uitvoeren van de software zonder dat dit de modulariteit van de implementatie schaadt, wat in de huidige programmeertalen niet mogelijk is. Daarom beschrijft dit proefschrift onze programmeertaal, genaamd Co-op, die het mogelijk maakt alle compositiemechanismen die nodig zijn voor het hergebruik van RUMs te implementeren.

Het combineren van onze ondersteuning voor RUMs tijdens zowel het ontwerp als de implementatie van software biedt softwareontwikkelaars de mogelijkheid energiegebruik modulair te optimaliseren. Dit vermindert het energiegebruik van software, terwijl het de software beter te onderhouden en begrijpen maakt. Hopelijk wordt onze aanpak in de praktijk toegepast voor het creëren van een duurzame wereld met software van hoge kwaliteit.

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

- B. Ploeger.** *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20
- T. Han.** *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21
- R. Li.** *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22
- J.H.P. Kwisthout.** *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23
- T.K. Cocx.** *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24
- A.I. Baars.** *Embedded Compilers.* Faculty of Science, UU. 2009-25
- M.A.C. Dekker.** *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26
- J.F.J. Laros.** *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27
- C.J. Boogerd.** *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01
- M.R. Neuhäuser.** *Model Checking Non-deterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02
- J. Endrullis.** *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03
- T. Staijen.** *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04
- Y. Wang.** *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05
- J.K. Berendsen.** *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06
- A. Nugroho.** *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07
- A. Silva.** *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08
- J.S. de Bruin.** *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09
- D. Costa.** *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10
- M.M. Jaghoori.** *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11
- R. Bakhshi.** *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01
- B.J. Arnoldus.** *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02
- E. Zambon.** *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Morali. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

- J. Wang.** *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25
- A. Khosravi.** *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01
- A. Middelkoop.** *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02
- Z. Hemel.** *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03
- T. Dimkov.** *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04
- S. Sedghi.** *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05
- F. Heidarian Dehkordi.** *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06
- K. Verbeek.** *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07
- D.E. Ndales Agut.** *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08
- H. Rahmani.** *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09
- S.D. Vermolen.** *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10
- L.J.P. Engelen.** *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11
- F.P.M. Stappers.** *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12
- W. Heijstek.** *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13
- C. Kop.** *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14
- A. Osaiweran.** *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15
- W. Kuijper.** *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16
- H. Beohar.** *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01
- G. Igna.** *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02
- E. Zambon.** *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03
- B. Lijnse.** *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points.* Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model.* Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development.* Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants.* Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking.* Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics.* Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes.* Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems.* Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities.* Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories.* Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond.* Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs.* Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage.* Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of*

ABCs with the Real World. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology.* Faculty of Sci-

ence, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16