Automatic Synthesis of Component & Connector-Software Architectures with Bounded Combinatory Logic

Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von
Boris Düdder

Dortmund
2014

Tag der mündlichen Prüfung:	25. August 2014
Dekan:	**Prof. Dr.-Ing. Gernot A. Fink**
Gutachter:	Prof. Dr. Jakob Rehof
	Prof. Ph.D. Fritz Henglein

Acknowledgment

Foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Jakob Rehof for the continuous support of my dissertation and research, for his inspiration, patience, motivation, enthusiasm, and profound knowledge. His guidance helped me in all the time of my research and of writing of this thesis. I could not have imagined having a better advisor and mentor for my research and dissertation.

Ditto, I would like to thank the rest of my promotions committee: Prof. Ph.D. Fritz Henglein, Prof. Dr. Bernhard Steffen, and Dr. Doris Schmedding, for their encouragement, insightful comments, and hard but constructive questions.

My sincere thanks also goes to Dr. Moritz Martens, Prof. Dr. Dietmar Jannach, Prof. Dr. Paweł Urzyczyn, Prof. Dr. Ugo de'Liguoro, Prof. Dr. Mariangiola Dezani-Ciancaglini, Zani Sarkisya, Anna Vasileva, Jan Bessai, and Andrej Dudenhefner, for working with me on various exciting projects and also to Ute Joschko for her help. Thanks are also due to my wife Olga, my brother Gordon, and my parents.

Abstract

Combinatory logic synthesis is a new type-based approach towards automatic synthesis of software from components in a repository. In this thesis we show how the type-based approach can naturally be used to exploit taxonomic conceptual structures in software architectures and component repositories to enable automatic composition and configuration of components, and also code generation, by associating taxonomic concepts to architectural building blocks such as, in particular, software connectors. Components of a repository are exposed for synthesis as typed combinators, where intersection types are used to represent concepts that specify intended usage and functionality of a component. An algorithm for solving the type inhabitation problem in combinatory logic — does there exist a composition of combinators with a given type? — is then used to automate the retrieval, composition, and configuration of suitable building blocks with respect to a goal specification.

Since type inhabitation has high computational complexity, heuristic optimizations for the inhabitation algorithm are essential for making the approach practical. We discuss particularly important (theoretical and pragmatic) optimization strategies and evaluate them by experiments. Furthermore, we apply this synthesis approach to define a method for software connector synthesis for realistic software architectures based on a type theoretic model. We conduct experiments with a rapid prototyping tool that employs this method on complex concrete ERP- and e-Commerce-systems and discuss the results.

Contents

1 Introduction **1**
- 1.1 Composition Synthesis and Inhabitation . . . 3
- 1.2 Combinatory Logic Synthesis . . . 5
- 1.3 Example for Combinatory Logic Synthesis . . . 7
- 1.4 Combinatory Logic Synthesis and Software Connectors . . . 9
- 1.5 Why does this problem matter? . . . 12
- 1.6 Synthesizing from Components . . . 13
- 1.7 Contributions . . . 13
  - 1.7.1 Publications & Delimitations . . . 13
  - 1.7.2 Theoretical Contributions . . . 15
  - 1.7.3 Technical Contributions . . . 15
  - 1.7.4 Interconnections between Contributions . . . 16
- 1.8 Organization . . . 17

2 Software Architecture **19**
- 2.1 What is Software Architecture? . . . 19
- 2.2 Describing Software Architectures . . . 21
  - 2.2.1 Components and Connectors . . . 22
  - 2.2.2 Software Connector Roles . . . 28
  - 2.2.3 Software Interconnection Models . . . 29
  - 2.2.4 Composing Basic Connectors . . . 30
- 2.3 Synthesizing Software Architectures . . . 31
- 2.4 Ontologies in Software Architecture . . . 31

3 Bounded Combinatory Logic **33**
- 3.1 Combinatory Logic . . . 33
- 3.2 Intersection Types . . . 35
  - 3.2.1 Types . . . 36
  - 3.2.2 Subtyping . . . 36
  - 3.2.3 Paths . . . 38
  - 3.2.4 Substitutions . . . 39

3.3 Bounded Combinatory Logic . . . 39
3.3.1 Type Assignment . . . 40
3.3.2 Relativized Inhabitation Problem . . . 41
3.4 Alternating Turing Machines . . . 43
3.5 Deciding Relativized Inhabitation . . . 44
3.6 Combinatory Logic Synthesis . . . 46
3.7 Related Work on Synthesis . . . 47

4 Optimization of CLS **51**
4.1 Theoretical Algorithm . . . 52
4.1.1 Restricting Intersections in Substitutions . . . 52
4.1.2 Intersection Type Matching . . . 56
4.1.3 Matching Optimization . . . 59
4.1.4 Lookahead Optimization . . . 61
4.1.5 Experimental Evaluation . . . 66
4.2 Algorithmic Optimization . . . 71
4.2.1 Execution Graph . . . 72
4.2.2 Term Level Optimization . . . 77
4.2.3 Elimination of Redundant Calculations . . . 80
4.2.4 Parallel Computation . . . 87
4.3 A Distributed Algorithm for $\mathsf{BCL}_0(\cap, \leq)$ Inhabitation . . . 94

5 Combinatory Logic Synthesizer **105**
5.1 Reconstructing Inhabitants . . . 105
5.2 Implementation . . . 106
5.3 (CL)S Input Specification . . . 110
5.4 (CL)S Output Specification . . . 112
5.5 Additional Applications and Extensions . . . 115

6 Synthesis of Software Architectures **119**
6.1 Software Connectors . . . 119
6.2 Type-theoretic Model . . . 120
6.2.1 Component . . . 120
6.2.2 Connector . . . 121
6.2.3 Building Blocks . . . 122
6.2.4 C&C Type Environment . . . 125
6.2.5 Taxonomy . . . 126
6.3 Combinatory Logic Connector Synthesis . . . 134
6.3.1 Synthesis of Connector . . . 134
6.3.2 Generation of a Connector . . . 135
6.3.3 Combinatory Logic Connector Synthesis Method . . . 136

6.3.4 Example for Combinatory Logic Connector Synthesis . 136
6.3.5 Synthesis of Behavior 138
6.3.6 Designing C&C Type Repositories and Templates . . . 139
6.4 Related Work 143

7 ArchiType **147**
7.1 UML2 Extension 148
7.2 User Interface 149
7.3 Synthesizing Connectors in UML2 150
7.4 Code Generation 152
7.5 ArchiType Composition Language 154
7.6 Staged Composition Synthesis 159
7.6.1 Implementation Type Correctness 161
7.7 Implementation 162
7.8 Related Work 164

8 Applications and Experiments **165**
8.1 Secure Connectors 166
8.1.1 Setup 166
8.1.2 Execution of the Experiment 166
8.1.3 Results 167
8.1.4 Analysis and Discussion 169
8.2 Detailed Broker Pattern Example 170
8.2.1 Setup 170
8.2.2 Execution 173
8.2.3 Results, Analysis, and Discussion 173
8.3 Enterprise Resource Planning Scenario 174
8.3.1 Setup 174
8.3.2 Execution 175
8.3.3 Results 179
8.4 e-Commerce Scenario 179
8.4.1 Setup 180
8.4.2 Execution 180
8.4.3 Results 188
8.4.4 Analysis and Discussion 188
8.5 Collective Analysis and Discussion 189
8.5.1 Expressiveness 189
8.5.2 Applicability 190
8.5.3 Adaptability 190
8.5.4 Costs and Effort 191
8.5.5 Usability 192

8.5.6 Limitations 193

9 Conclusion **197**

A Selected implementation details **199**
A.1 Grammar of (CL)S's input language 199
A.2 Example in the Listing in ArchiType CL 201
A.3 Example Template in T4 202
A.4 Algorithm Listings 205
A.4.1 Inhabitation Algorithm (data-centric part) 205

B Experiments **207**
B.1 Compiler 207
B.1.1 Compiler Flags 207
B.2 Benchmark Problems 207
B.3 Configuration of the Test Systems 209
B.3.1 Test System I - Desktop PC 209
B.3.2 Test System II - Compute Server 209

Acronyms **211**

List of Figures **215**

List of Tables **219**

List of Algorithms **221**

Bibliography **223**

Index **241**

Chapter 1

Introduction

Software systems tend to be large and complex. These factors must be managed in the design and development of such systems. Except for product lines, software architectures are unique products. Design and development take the form of a project and the development process is repeated for every software system that is constructed.

Over the years, software engineers designed and developed software applications and -systems with increasing complexity. Different techniques have been developed to handle growing complexity of systems. A very powerful tool is the abstraction of a system's building blocks. The very low-level abstraction of functions in procedural programming was replaced by objects in object-oriented programming. From there, objects have been abstracted to distributed programming and after that to component-based programming. The idea of component-based programming is to divide software applications and -systems into reusable components. The advantage of reusable components is that the number of newly created components for a new application can be reduced by reusing existing components. That means that components only have to be developed for missing functionality that is not covered by existing components.

The idea of component marketplaces emerged where components are traded. Furthermore, topically coherent components have been bundled into libraries or repositories in order to increase revenue for producers of such repositories and also to increase additional benefit for users of component repositories.

In order to develop a component-based software application or -system suitable components have to be retrieved from a suitable component repository and composed or assembled in a way such that the composite application fulfills the functional and non-functional requirements demanded by the end-user.

As it turned out, it was not that easy to compose components in a goal-directed fashion. Components are typically documented in textual form. The sheer quantity of components that have to be used to compose the intended software application made such an approach intricate and error-prone. From this problem, the idea emerged to automate composition by synthesizing a composition plan describing which components have to be composed with other components to reach a specific synthesis goal. *Component-oriented synthesis* is the goal-directed construction of a composite software system composed by components residing in a component repository.

On an even higher level of abstraction, the production cost of software systems could be reduced through a higher degree of reuse of architectural elements. This is not that easy to achieve, because architectures contain interactions among elements as well as computations in specific elements satisfying the user's functional and non-functional requirements. Reuse is hindered by varying, especially functional, requirements some of which may be unique to the software system. Therefore, synthesis of whole software systems is desirable but not easily achievable.

The distinction between interaction and computation comes naturally. Thus, a distinction between architectural elements seems reasonable. Software components are the architectural elements in which computation resides and software connectors are responsible for interactions among those components.

The functionality of a software system stems from its computation. We know that the computational part may not be easily synthesizable and reuse does not bring much advantage because the computational part is likely unique for every software system. On the other hand, the interaction of components via software connectors is somewhat different. Composable connectors can be reused for different software systems. The versatility of connectors can be achieved by the composition of connector building blocks that create the desired functionality.

Another motivation for focusing on software connectors is that a software architect's knowledge can be exploited. Such a knowledge can be represented formally as a taxonomy and then be used by algorithms for reasoning with this taxonomy. Existing work on software connector's taxonomies can also be used and also be further developed. This approach requires that the synthesis method also exploits taxonomic information.

In this thesis we study a novel approach to component-oriented synthesis which is based on combinatory logic. In the remainder of this chapter we will illustrate this approach in broad outline.

1.1 Composition Synthesis and Inhabitation

In general, composition synthesis is a method for synthesizing compositions of typed functions which is based on combinatory logic [Rehof, 2013]. One can broadly classify this approach within the line of work often referred to as deductive program synthesis.[1] However, whereas deductive program synthesis has traditionally been based on Hoare-style program logics and have been pursued within semi-automated frameworks, composition synthesis is founded on type theory and aims at fully automated synthesis. Moreover, whereas synthesis has traditionally been pursued as the construction of a program or a system *from scratch*, combinatory logic synthesis is a component-oriented approach to synthesis, in which synthesis is *relativized* to arbitrary libraries (repositories) of components. Component-oriented synthesis is a surprisingly recent development[2] which is also being pursued in the technically very different setting of temporal logic and automata theory [Lustig and Vardi, 2009].

In its minimal form, composition synthesis only consists of a single logical rule(referred to as implication elimination, modus ponens, or application):

$$\frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ e') : \tau}(\to\mathrm{E})$$

Here, Γ is a *combinatory type environment* consisting of bindings of the form (X, ρ) where X is a combinator symbol and ρ is a type. Applicative expressions (ee') are built from combinator symbols and are typed by the rule shown above. This rule can be used to constitute the simplest and most fundamental logical model of applicative composition of *named component specifications*, where Γ models a repository of *named components* (X) and their type specifications (ρ). The basic idea behind combinatory logic synthesis is to consider the component-oriented synthesis problem as being modeled by the problem of combinatory type *inhabitation*. For fixed Γ and τ as input, the *inhabitation problem* is the following decision problem

$$\exists e.\ \Gamma \vdash e : \tau?$$

or, does there exist a composition e of components from repository Γ satisfying goal τ,

$$\Gamma \vdash e : \tau?$$

[1] Deductive program synthesis has been developed by Manna and Waldinger [1980] and later more refined by Waldinger [1990]. Deductive program synthesis founded the base for a family of deductive synthesis approaches like the synthesis approach of Traugott [1989] on deductive synthesis of sorting programs. Manna and Waldinger [1992] present a comprehensive tutorial on deductive program synthesis.

[2] `http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=14232`

An (inhabitation) algorithm is used to *construct* or synthesize a composition e from Γ and τ. The inhabitation algorithm is the foundation for automatic synthesis and is inherently component-oriented, because combinatory environments only expose component names and their types to synthesis. Later on, we will use the abbreviating notation $\Gamma \vdash ? : \tau$ for the inhabitation problem.

In the type-based approach to composition synthesis, types (τ) take the role of specifications of named components represented by terms (e):

$$\text{Types} \quad \tau, \tau' \quad ::= \quad a \mid \alpha \mid \tau \to \tau'$$

$$\text{Terms} \quad e, e' \quad ::= \quad X \mid (e\, e')$$

Types are constructed by using type constants (a), type variables (α), and function types ($\tau \to \tau'$). Terms are constructed by using component names or combinators X and using application of e to e', (ee'). In standard combinatory logic [Hindley and Seldin, 2008], an additional rule (var) is added to allow schematic instantiation of combinator types under substitutions S.

$$\frac{[\text{Substitution } S]}{\Gamma, (X : \tau) \vdash X : S(\tau)}(\text{var})$$

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\, e') : \tau}(\to \text{E})$$

From a type-theoretic point of view, combinator types are considered as being *implicitly polymorphic*.[3] From a logical point of view, under the Curry-Howard isomorphism [Sørensen and Urzyczyn, 2006], the system is a Hilbert-style presentation of minimal implicative propositional logic. Viewed thus, the environment (repository) Γ represents a propositional theory, and inhabitation is the question of provability for this theory. Famously, if Γ is the following fixed base, then the combinatory logic (**SK**) is equivalent to the full λ-calculus:

$$\{\mathbf{S} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \mathbf{K} : \alpha \to \beta \to \alpha\}$$

The inhabitation problem in **SK**-calculus is shown PSPACE-complete by Statman [1979].

But a fixed base is *not* the right model for composition synthesis, since repositories (Γ) *vary*, and, moreover, λ-calculus as a model is not *component-oriented* as is combinatory logic. Combinatory logic synthesis must therefore

[3] Also sometimes referred to as *typical ambiguity*.

be based on the *relativized* inhabitation problem where the input theory Γ is not held fixed but is given as part of the input:

Given Γ and τ as input, does there exist e such that $\Gamma \vdash e : \tau$

It turns out [Rehof, 2013] that the relativized inhabitation relation is surprisingly expressive: the relativized inhabitation problem is undecidable, already in simple types (Linial-Post theorems [Linial and Post, 1949]). As detailed in [Rehof, 2013], relativized inhabitation can be seen as defining the semantics for a Turing-complete logic programming language for computing type-correct compositions. Under this viewpoint, Γ can been regarded as a logic program, typed combinators $(X : \tau') \in \Gamma$ are its rules, τ its input goal, and search for inhabitants its execution semantics.

1.2 Combinatory Logic Synthesis

Combinatory logic synthesis (CLS) was proposed by Rehof [2013] as a type-based approach to synthesis from components in a repository. Based on combinatory logic introduced by Schönfinkel (1923) and again by Curry (1927), components are specified semantically in CLS by intersection types introduced in [Barendregt et al., 1983]. Hindley and Seldin [2008] provide a synopsis on combinators and combinatory logic.

Combinatory logic synthesis refines composition synthesis by augmenting the specification of components with semantic information encoded by intersection types. We briefly sketch the outline of the formal system underlying CLS. Full details of the system will be given in Chapter 3 on page 33.

$$\text{Semantic types} \qquad \mathtt{t} \quad ::= \quad \mathtt{a} \mid \mathtt{t} \to \mathtt{t}' \mid \mathtt{t} \cap \mathtt{t}'$$

$$\text{Semantically annotated types} \qquad \phi \quad ::= \quad \tau \mid \phi \cap \mathtt{t} \mid \phi \to \phi' \mid \phi \cap \phi'$$

The intersection type operator ($\cap$) was introduced by Dezani-Ciancaglini [Barendregt et al., 1983] and was integrated into combinatory logic in [Dezani-Ciancaglini and Hindley, 1992]. We note semantic types in blue color. Additionally, two extra type rules ($\cap$I and $\leq$) are added.

$$\frac{[\text{Substitution } S]}{\mathcal{C}, X : \phi \vdash X : S(\phi)}(\text{var}) \qquad \frac{\mathcal{C} \vdash e : \phi \to \phi' \quad \mathcal{C} \vdash e' : \phi}{\mathcal{C} \vdash (e\ e') : \phi'}(\to\text{E})$$

$$\frac{\mathcal{C} \vdash e : \phi \quad \mathcal{C} \vdash e : \phi'}{\mathcal{C} \vdash e : \phi \cap \phi'}(\cap\text{I}) \qquad \frac{\mathcal{C} \vdash e : \phi \quad \phi \leq \phi'}{\mathcal{C} \vdash e : \phi'}(\leq)$$

The intersection type operator, $\cap$, captures semantic information by adding semantic types ($\mathtt{t}$) to a type (ϕ) resulting in semantically annotated types ($\phi \cap \mathtt{t}$). Rule ($\leq$) adds subtyping and can be used to encode taxonomic hierarchies. In the next subsection, an example will demonstrate the idea of using intersections to specify components in a repository and synthesizing compositions in more detail.

The situation in combinatory logic synthesis can be briefly summarized as follows: We are *given* a repository (set) of component names $\mathtt{X_i}$ with associated implementations or combinators C_i of type τ_i, e.g. INT of Java, in a native implementation language $\mathsf{L1}$, e.g. Java or ML,

$$\mathtt{X_1} \triangleq C_1 : \tau_1, \ldots, \mathtt{X_n} \triangleq C_n : \tau_n$$

A name $\mathtt{X_i}$ is used as label or placeholder for a concrete implementation C_i in $\mathsf{L1}$. In addition, an associated repository forming a combinatory type environment

$$\mathcal{C} = \{\mathtt{X_1} : \phi_1, \ldots, \mathtt{X_n} : \phi_n\}$$

is given where ϕ_i are enriched types with $(\phi_i)^\circ \equiv \tau_i$. Enriched types ϕ_i with $i \in \{1, \ldots, n\}$ are projected by erasure $(\cdot)^\circ$ to a corresponding implementation language type. Enriched types ϕ_i are augmented by adding semantic information to τ_i describing the type of the implementation of C_i in a native implementation language $\mathsf{L1}$. Then, using inhabitation, we *ask for* combinatory compositions e such that

$$\mathcal{C} \vdash e : \phi \quad \text{and} \quad \vdash_{\mathsf{L1}} e[C_i/\mathtt{X_i}] : (\phi)^\circ$$

Here, the condition $\vdash_{\mathsf{L1}} e[C_i/\mathtt{X_i}] : (\phi)^\circ$ is *implementation type correctness* and guarantees that combinatory compositions e are well-typed programs in the given native implementation language $\mathsf{L1}$ after substituting all occurring labels $\mathtt{X_i}$ with their corresponding implementations C_i.[4]

It should be noted that CLS considers semantic types as given and is not per se concerned with establishing their correctness by type checking semantic types against component implementations. Doing so is considered an orthogonal issue. In this regard, the CLS approach follows work on adaptation synthesis via proof counting by Haack et al. [2002] as well as Wells and Yakobowski [2005], where semantic types are combined with proof search in a specialized proof system.

[4] We refer to Section 7.6.1 on page 161 for a more detailed discussion about implementation type correctness in the thesis.

1.3 Example for Combinatory Logic Synthesis

Rehof [2013] presented the following example that is used to exemplify the idea of combinatory logic synthesis.

Example 1.1. *We assume an implemented repository of components or application programming interface* (API) *providing functionality for tracking temperature-sensitive containers, e.g. reefer containers, in logistics. The following repository* Γ *represents the existing (API-)functions:* `R` *is the data-type real. Function* `0` *returns a tracking object* `TrObj`*. Given a tracking object* `TrObj`*, function* `Tr` *returns a triple whose first entry is the coordinate of the tracking object, followed by the time information, and the current temperature of the tracking object. The function* `pos` *projects a position and time from such a triple. Function* `cdn` *projects a coordinate and functions* `fst` *and* `snd` *project the first respectively second entry in a pair like a coordinate. Function* `tmp` *returns the temperature. Two additional conversion functions* `cc2pl` *and* `cl2fh` *are contained, that convert Cartesian to polar coordinates and respectively temperature from Celsius to Fahrenheit.*

$\Gamma = \{$

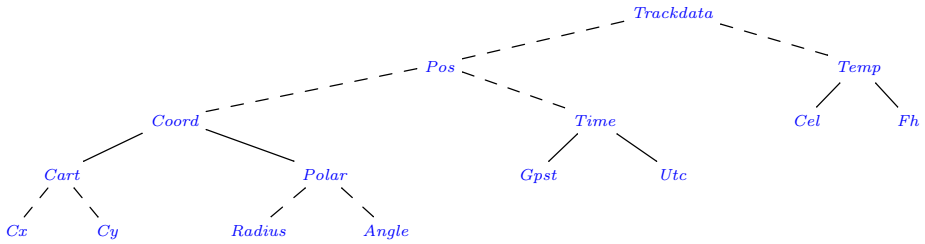
$$
\begin{array}{lcl}
\texttt{0} & : & \texttt{TrObj}, \\
\texttt{Tr} & : & \texttt{TrObj} \to \texttt{D}((\texttt{R},\texttt{R}),\texttt{R},\texttt{R}), \\
\texttt{pos} & : & \texttt{D}((\texttt{R},\texttt{R}),\texttt{R},\texttt{R}) \to ((\texttt{R},\texttt{R}),\texttt{R}), \\
\texttt{cdn} & : & ((\texttt{R},\texttt{R}),\texttt{R}) \to (\texttt{R},\texttt{R}), \\
\texttt{fst} & : & (\texttt{R},\texttt{R}) \to \texttt{R}, \\
\texttt{snd} & : & (\texttt{R},\texttt{R}) \to \texttt{R}, \\
\texttt{tmp} & : & \texttt{D}((\texttt{R},\texttt{R}),\texttt{R},\texttt{R}) \to \texttt{R}, \\
\texttt{cc2pl} & : & ((\texttt{R},\texttt{R}),\texttt{R}) \to ((\texttt{R},\texttt{R}),\texttt{R}), \\
\texttt{cl2fh} & : & \texttt{R} \to \texttt{R}
\end{array}
$$

$\}$

In the native API shown, however, the semantic information given above cannot be represented. Semantic information is typically only included, if at all, in the form of informal text describing the repository as accompanying documentation, which cannot be exploited by formal or automatic methods.

In order to get formal semantic structure representing such information, a taxonomic hierarchy can be specified. Dashed lines denote a has-a *relationship whereas continuous lines denote an* is-a *relationship. We will denote concepts in blue again. Trackdata contains a position Pos and a temperature Temp. A temperature can be measured in units of Celsius Cel or Fh. A coordinate can be represented in Cartesian Cart or polar Polar coordinates. A Cartesian*

coordinate is a pair of x Cx *and y* Cy *coordinates whereas a polar coordinate is a pair of radius* $Radius$ *and angle* $Angle$*.*



The taxonomic hierarchy and intersection types $\cap$ *can be used to superimpose semantic concepts onto the native API, thereby refining the specification of functions contained in the repository* Γ *with semantic information:*

$\mathcal{C} = \{$

$$
\begin{array}{lcl}
\texttt{O} & : & \texttt{TrObj}, \\
\texttt{Tr} & : & \texttt{TrObj} \to \texttt{D}((\texttt{R},\texttt{R}) \cap Cart, \texttt{R} \cap Gpst, \texttt{R} \cap Cel), \\
\texttt{pos} & : & \texttt{D}((\texttt{R},\texttt{R}) \cap \mathfrak{a}, \texttt{R} \cap \mathfrak{a}', \texttt{R}) \to ((\texttt{R},\texttt{R}) \cap \mathfrak{a}, \texttt{R} \cap \mathfrak{a}') \cap Pos, \\
\texttt{cdn} & : & ((\texttt{R},\texttt{R}) \cap \mathfrak{a}, \texttt{R}) \cap Pos \to (\texttt{R},\texttt{R}) \cap \mathfrak{a}, \\
\texttt{fst} & : & ((\texttt{R},\texttt{R}) \cap Coord \to \texttt{R}) \cap \\
 & & (Cart \to Cx) \cap (Polar \to Radius), \\
\texttt{snd} & : & ((\texttt{R},\texttt{R}) \cap Coord \to \texttt{R}) \cap \\
 & & (Cart \to Cy) \cap (Polar \to Angle), \\
\texttt{tmp} & : & \texttt{D}((\texttt{R},\texttt{R}), \texttt{R}, \texttt{R} \cap \mathfrak{a}) \to \texttt{R} \cap \mathfrak{a}, \\
\texttt{cc2pl} & : & (\texttt{R},\texttt{R}) \cap Cart \to (\texttt{R},\texttt{R}) \cap Polar, \\
\texttt{cl2fh} & : & \texttt{R} \cap Cel \to \texttt{R} \cap Fh
\end{array}
$$

$\}$

Note, in particular, the usage of the intersection-operator ($\cap$) *to combine several specifications into a single type, e.g. in combinator* `cl2fh` *the intersection type* $\texttt{R} \cap Fh$*. In* $\mathcal{C}$*,* is-a *relations are translated to intersections with semantic types and* has-a *relations are expressed by projection combinators like* `tmp`*. The repository reflects the semantic information that has been provided in textual form in the description of the implementation repository.*

We can now use relativized type inhabitation to ask for synthesizing a composition being a function that returns a temperature in Fahrenheit, $\texttt{R} \cap Fh$*.*

$$\mathcal{C} \vdash ? : \texttt{R} \cap Fh \rightsquigarrow \mathcal{C} \vdash \texttt{cl2fh (tmp (Tr O))} : \texttt{R} \cap Fh$$

Constructing an inhabitant, denoted by $\rightsquigarrow$*, satisfying the (type-)specification* $\texttt{R} \cap Fh$ *leads to the term* `cl2fh (tmp (Tr O))` *as solution.*

$$\mathcal{C} \vdash ? : \mathtt{R} \cap \mathit{Radius} \rightsquigarrow \mathcal{C} \vdash \mathtt{fst\ (cc2pl\ (cdn\ (pos\ (Tr\ 0))))} : \mathtt{R} \cap \mathit{Radius}$$

A more complex inhabitant is constructed for the inhabitation request for a function returning Radius as a real. The inhabitant

```
fst (cc2pl (cdn (pos (Tr 0))))
```

satisfies the inhabitation request R∩*Radius*.

We return to synthesis of software connectors using combinatory logic synthesis.

1.4 Combinatory Logic Synthesis and Software Connectors

The type-based approach is well suited for associating semantic specifications with components in a natural way, since components exhibit type information through their interfaces. In the software engineering community, taxonomies [Hirsch et al., 1999, Mehta et al., 2000] and ontologies [Kruchten, 2004] have been introduced and examined to structure and reason about classes of architectural elements such as, for instance, software connectors.

In the thesis we show how the type-based approach to synthesis can be used to exploit such information in order to automate composition and configuration of components, code generation, and deployment instructions. In the methodology proposed here, software architectural concepts organized in a taxonomy are used to specify the intended usage and other relevant properties of components. Such concepts are attached to components via type structure (for example, component interfaces), and are regarded as *semantic* types following the ideas of Haack et al. [2002] and Wells and Yakobowski [2005]. Taxonomic relations between concepts in taxonomic hierarchies can technically be represented as subtyping partial orders. To semantically describe components we use intersection types by Barendregt et al. [1983] to append concepts to component types. The type-based approach results in an intuitive form of specification.

Technically, the basic idea in CLS is to represent a repository of components as a combinatory type environment Γ [Rehof and Urzyczyn, 2011a, Rehof, 2013], that is, a set of type assumptions of the form $(x : \tau)$, where x is the name of a component regarded as a combinator symbol and τ is its semantic type. Informally, the statement $(x : \tau \cap \sigma)$, where $\tau \cap \sigma$ is an

intersection type, means that x has both type τ and type σ, i.e., x satisfies both specifications. For example, in the following specification in a scenario where software components have required and provided interfaces

$$x : (I_1 \to I_2) \cap ws \cap sec$$

the combinator symbol x names a connector connecting a component providing interface I_1 to a component requiring interface I_2. The type $(I_1 \to I_2)$ is enriched with semantic types *ws* and *sec* by means of intersections, expressing the fact that x is a secure web service.

Type inhabitation is the decision problem: given a type environment Γ and a type τ, does there exist a combinatory term (applicative combination of combinator symbols) e such that $\Gamma \vdash e : \tau$ holds. $\Gamma \vdash e : \tau$ states that combinatory term e has type τ in the type environment Γ. An algorithm for type inhabitation solves the problem of *component identification*, *retrieval*, and *composition* and results in a combinatory term, e, specifying how components are to be composed to achieve a certain goal specification, τ.

Besides the semantic specification each component has an underlying implementation that is hidden from the user. Such implementations may consist of code templates or meta-code describing how code templates are to be composed. These templates are used for the code generation. Composing the underlying templates according to the combinatory term e results in compilable code. We say that e is *synthesized* from the repository Γ and that code is *generated* from e. The theoretical foundation of CLS has been laid by recent research on the inhabitation problem in finite and bounded combinatory logic by Rehof and Urzyczyn [2011a], Rehof and Urzyczyn [2012], Düdder et al. [2012], Rehof [2013], Düdder et al. [2012], and Düdder et al. [2013a].

The basis theory of CLS having been developed recently, the methodology still requires investigation regarding its applicability in practice. Theoretical results by Rehof and Urzyczyn [2011a, 2012], Düdder et al. [2012] show that the inhabitation problem has high theoretical complexity: for the logic used here, bounded combinatory logic denoted by $\mathsf{BCL}_k(\cap, \leq)$, type inhabitation is shown to be $(k+2)-$EXPTIME-complete in [Düdder et al., 2012]. Thus, CLS requires heuristic optimizations to be useful in practice. Based on a solution to the intersection type matching problem presented in [Düdder et al., 2013a], we present, in the first part of the thesis, a heuristically optimized version of the theoretical inhabitation algorithm of Düdder et al. [2012] that provides significant speedups. We discuss experimental data obtained from applying the optimized algorithm to an example. The example tests different aspects of that algorithm and a speedup of several orders of magnitude in the

optimized algorithm compared to the theoretical algorithm can be witnessed. Besides pinpointing the reasons for a speedup we also identify problems that the optimized algorithm has to cope with. From these observations we derive design principles for the specification of repositories. After the optimization of the theoretical algorithm, a variety of practical optimizations that are relevant for an implementation and an implemented software tool named Combinatory Logic Synthesizer (CL)S containing all optimizations for the theoretical and the practical algorithm are presented.

In the second part of the thesis, we present two realistic, larger-scale applications of (CL)S to software connector synthesis [Perry and Wolf, 1992, Allen and Garlan, 1997, Taylor et al., 2010]. Connector construction typically requires the repeated solution of basically similar but varying tasks, regarding both configuration and coding. This is usually a complex process, since there can be a huge number of such variations, and managing these can be tedious and error-prone if done manually. Making it difficult to reuse already created connectors. Furthermore, varying frameworks, technologies, and implementation languages require an architect to have in-depth technical knowledge that cannot always be expected. To solve this problem, Spitznagel and Garlan [2001, 2003] propose composite connectors composed from building blocks. However, this compositional approach still requires the retrieval and composition of suitable building blocks. Hirsch et al. [1999] and also Mehta et al. [2000] describe properties of connectors and building blocks by comprehensive taxonomies. Such taxonomies reflect best-practice knowledge in software architecture. Thus, they may help optimize the connector-synthesis for specific contexts. We present a type-theoretic model describing connectors and building blocks as well as connector taxonomies.

We implemented a tool, ArchiType, which fully automates software connector synthesis based on (CL)S. In ArchiType synthesized combinatory terms are interpreted in an *architecture description language* (ADL) from which compilable code is generated. Since types are abstract, there is no need to deal with actual implementations, code, or configurations. Moreover, the approach is agnostic of the ADL and target technology used.

We used ArchiType to synthesize and generate connectors to transform two monolithic real-world software-systems into distributed software systems. A mid-sized open-source *enterprise resource planning* (ERP) system comprising of 38 250 *logical lines of code* (LLOC) and a larger-sized eCommerce-application comprising of 225 763 LLOC are used for experiments and demonstration of the method proposed in the thesis. At the end, we deployed the distributed architectures on different virtual machines to verify that indeed functionally correct code was generated.

1.5 Why does this problem matter?

Complex software systems consist of thousands of interdependent components that use many interconnections to meet functional and non-functional requirements. The number of system interconnections can be huge.[5] Furthermore, the interconnections affect the software system's functional and also non-functional properties. Moreover, these interconnections depend on different aspects like the technology used. A software architect usually constructs prototypical implementations of his software architecture for testing and evaluating his ideas. But, modifying, changing, or adding software connectors in the software architecture is time-consuming and therefore the evaluations are only conducted for a few, well-chosen variants of the software architecture.

Furthermore, because of system complexity and emerging risks, a software architect is obliged to construct prototypical implementations to experimentally evaluate the system's functional and non-functional properties. The business logic (operations) is located in the system's components. These components have to be implemented together with a possibly large (in the worst-case quadratic) number of connectors. Hence, the number of interconnections has a strong impact on the time needed for the prototype's construction. Probably, we cannot avoid the construction of software connectors in order to reduce the needed construction time. However, we can reduce the time that is spent on the construction of *each* connector. Therefore, the automatic synthesis *and* code generation could accelerate the development and the evaluation of software architectures.

Then, automatic synthesis can be applied to different fields of application. A rapid prototyping tool for software architects can synthesize prototypical software architectures, e.g. for evaluating various properties and aspects of architectural design choices. By supporting different middleware systems and communication infrastructures, the initial skill adaption training and search in technological software and documentation libraries can be reduced. Another field is the application in software product lines. Software product lines, vary software products by including or excluding features depending on the actual product line.

In addition, the thesis is closely related to actual movements[6] in the component-based synthesis community. In general, synthesis approaches can be classified based on a respective model and related methods. Two distinct and often independent approaches can be distinguished, sometimes called "Church-style" synthesis and "Curry-style" synthesis, respectively. Church-

[5] In the e-Commerce example in Chapter 8 on page 165, 400 components depend on a central component.

[6] `http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=14232`

style synthesis is characterized by the usage of temporal logic and automata theoretic models, whereas Curry-style synthesis is characterized by the usage of deductive methods in program logics and in type theory. Recent work, for example by Lustig and Vardi [2009], has inspired the idea of component-based synthesis, where systems are synthesized relative to a given collection (library, repository) of components within both the Church-style and the Curry-style approach. Delineating a collection needs included components to be designed for composition requiring a non-trivial degree of design intelligence, abstraction, and specification. The thesis can be classified as Curry-style component-based synthesis for software connectors as an application domain providing such a non-trivial design intelligence, abstraction, and specification for software architecture as a domain.

1.6 Synthesizing from Components

Today's software developers and architects can use components from off-the-shelf component libraries originating from the idea of a global component marketplace favored by component-based development. In contrast, a software architect cannot choose the software connectors from an analogous library. Such a lack is a serious deficit for software architects. Particularly, various communication frameworks like the Eclipse Communication Framework or the Microsoft *windows communication foundation* (WCF) try to close the gap with their service offerings. However, a software connector fulfills more than a communication role. Hence, one major contribution of the thesis is to remedy these deficiencies by providing a synthesis and generation method as well as providing the idea and a concrete implementation of a software connector library.

1.7 Contributions

The contributions of the thesis are twofold and can be categorized into theoretical and technical contributions. Before the contributions are listed, the primary publications, their respective contributions, and a delimitation to the original and proprietary contributions of the thesis are presented.

1.7.1 Publications & Delimitations

The following list of co-authored peer-reviewed publications contribute and influence the thesis.

- The paper "Using Inhabitation in Bounded Combinatory Logic with Intersection Types for GUI Synthesis" by Düdder et al. [2012] presents some applications combinatory logic synthesis to various scenarios.

- The paper "Bounded Combinatory Logic" by Düdder et al. [2012] provides the theoretical foundation, complexity results, and algorithms for bounded combinatory logic. The results are not repeated in this work but are essential for the thesis.

- The paper "Intersection Type Matching with Subtyping" by Düdder et al. [2013a] presents a complexity analysis and an algorithm for type matching in intersection types based on the algorithm presented in [Düdder et al., 2012]. In addition, an optimized algorithm for deciding relativized type inhabitation is presented that is similar to the Algorithm 4.5 on page 64. Algorithm 4.5 is an accumulation of the preceding optimizations presented in Section 4.1. In addition, various algorithmic optimizations are added in the thesis and result in an implementation of the algorithm used by a tool called (CL)S in Chapter 5.

- The paper "Staged Computation Synthesis" by Düdder et al. [2014a] introduces a meta-language into composition synthesis by adding the modal type constructor $\Box$ to distinguish native language and meta-language in synthesis. This approach, is similar to the approach presented in Section 7.5 on page 154 that presents the ArchiType Computation Language corresponding to the meta-language presented in Düdder et al. [2014a].

- The paper "Delegation-based Mixin Composition Synthesis" by Bessai et al. [2014a] shows an application of combinatory logic synthesis for object-oriented software systems with a focus on synthesizing compositions of mixins.

- The paper "Combinatory Logic Synthesizer" by Bessai et al. [2014b] provides an overview of recent, current, and future features of the Combinatory Logic Synthesizer (CL)S that has been developed in the thesis.

- The paper "Model Checking in multiagentengesteuerten Materialflusssystemen" by Düdder et al. [2008] and the diploma thesis "Formale Verifikation mittels Model Checking in Materialflusssystemen" by Düdder [2008] are not related to this thesis.

1.7.2 Theoretical Contributions

The thesis provides the following theoretical contributions

- First, a combinatory logic for specifying components is presented.
- Second, for this logic, an approach for combinatory logic synthesis using type inhabitation is presented.
- Third, the complexity results presented in Chapter 3 on page 33 for the type inhabitation algorithms for the combinatory synthesis necessitate heuristical optimizations to enable the practical usability of the combinatory synthesis for concrete scenarios. Hence, different improvements and heuristical optimizations are discussed and evaluated experimentally.
- Fourth, *Combinatory Logic Connector Synthesis*, a type-based method for the synthesis of software connectors in component & connector (C&C) architectures is presented. This method offers
  - a mapping of an abstract context and problem domain of specific software architectures into a concise specification in combinatory logic preserving the domain's peculiar semantic.
  - an efficiently[7] computable synthesis of software connectors exploiting the specifications using the combinatory logic.
  - a mapping from the synthesis results into a description language for software architectures.
  - a goal-oriented synthesis and generation of software connectors that provide syntactic *and* semantic interconnections.

1.7.3 Technical Contributions

The technical contributions supplement the theoretical contributions and can be summarized as follows

- Fifth, a variety of pragmatic optimizations for combinatory logic synthesis with respect to a practical implementation of the synthesis algorithm is introduced. The implementation takes advantage of the theoretical contributions.

[7] Efficient not from the a priori runtime complexity but from an algorithm comprising the optimizations.

- Sixth, an implementation of the combinatory logic synthesis algorithm including the mentioned optimizations is presented. Experimental results give evidence for the significant improvements that these optimizations provide.

- Seventh, a rapid prototyping software architecture tool using the implementation of the combinatory logic synthesis algorithm and the Combinatory Logic Connector Synthesis method is discussed.

- Eighth, the rapid prototyping software architecture tool is used to conduct four experiments on *real* software systems to verify the benefits of the Combinatory Logic Connector Synthesis method in the presence of real-world software architectures.

1.7.4 Interconnections between Contributions

Main contributions included in the thesis are interconnected in the following way as depicted in Figure 1.1

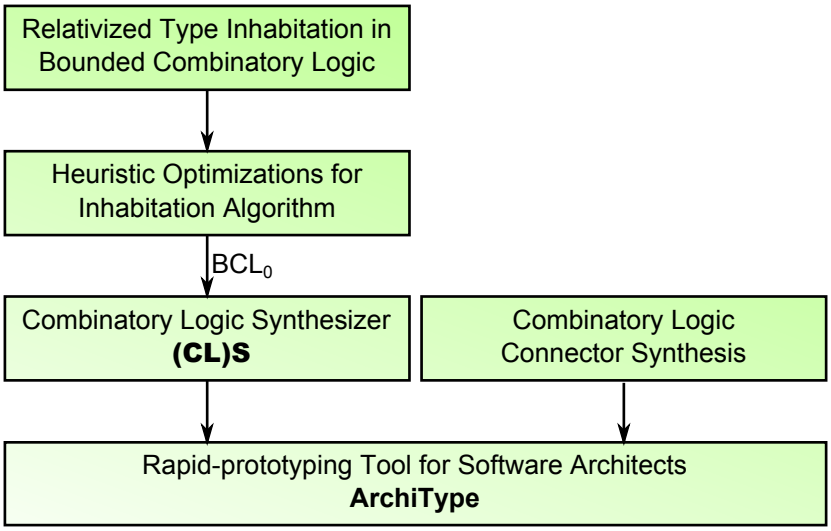


Figure 1.1: Interconnections between main contributions in the thesis

- Relativized type inhabitation, a type-theoretic decision problem, in bounded combinatory logic will be presented and used as synthesis procedure.

- Various heuristics yield an optimized synthesis algorithm.

- Combinatory Logic Synthesizer (CL)S, a tool, encapsulates the synthesis algorithm in its core and provides an abstract interface for synthesis.
- Type-theoretic model of software components and connectors is a requirement for Combinatory Logic Connector Synthesis allowing synthesis of software connectors.
- Thereby, (CL)S is employed for a rapid-prototyping tool for software architects, ArchiType, implementing Combinatory Logic Connector Synthesis.

1.8 Organization

The thesis is structured in the following way

- Chapter 1 serves as general introduction, sets the context of the thesis and motivates the problem. Furthermore, it outlines the organization of the thesis and describes the contributions of this work.
- Chapter 2 introduces problems in software architecture and its methods. The general problem of the synthesis of architectural elements and related work is discussed. The discussion leads to the (general) main idea of the thesis which will be concretized in the remainder of the thesis.
- Chapter 3 contains the theoretical foundations of the thesis. Here, the combinatory logic as well as the synthesis algorithm on which the thesis is based are defined. Both main artifacts (logic and synthesis) are put into context with related work.
- Chapter 4 presents different theoretical and also pragmatic optimization approaches for the synthesis algorithm. The optimizations are derived by analyzing the algorithm(s) and developed further from type-theoretical to pragmatic optimizations.
- Chapter 5 presents an implementation called (CL)S of the synthesis algorithm including the theoretical and pragmatic optimizations.
- Chapter 6 precisely defines the method Combinatory Logic Connector Synthesis, provides a classification system for software connectors, and describes the usage of combinatory logic synthesis for synthesizing compositions of software connectors.

- Chapter 7 reviews a rapid prototyping software architecture tool (ArchiType) using (CL)S and that is integrated in an industrial software engineering tool providing the Combinatory Logic Connector Synthesis method to a software architect. The method is extended by including a model-to-code transformer which generates compilable source code that implements the synthesized software connector with an user-specified functionality.

- Chapter 8 evaluates the Combinatory Logic Connector Synthesis method with the help of experiments in which the rapid prototyping software architecture tool is applied to different software systems with increasing complexity. The experiment's consolidated findings and lessons learned are also included in the description of the individual experiments.

- Chapter 9 concludes the thesis by a résumé and a perspective on future work.

- The appendices provide more detailed information on specific technical details, of the tools as well as the conducted experiments and their results.

- The discussion on related work with respect to this thesis is split into two separate discussions in Chapters 3 and 6. This separation is reflects the fact that this thesis merges results coming from two separate scientific communities, type theory/logic and software architecture.

Chapter 2

Software Architecture

In this chapter we will discuss various and differing definitions of software architecture. From this discussion we will derive definitions of important architectural entities like software components and software connectors. Both architectural elements will be studied separately in order to assess their various properties and features. With synthesis in mind, we will discuss composition and synthesis of architectural elements and argue for the need of semantic specifications of the interconnections of architectural elements.

2.1 What is Software Architecture?

Software architecture is a sub discipline of software engineering. The definition of software architecture depends largely on specific authors and their respective intentions. Some of the more influential definitions for software architecture are discussed in the following.

The IEEE-Standard 1471-2000 [IEEE Architecture Working Group, 2000, page 9] defines software architecture as:

Definition 2.1.1. ***(Software architecture I)***
"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

The definition primarily considers software intensive systems and explicitly considers the evolution of these systems. Shaw and Garlan [Shaw and Garlan, 1996] promoted the idea of software architecture concepts such as components, connectors, and styles: "The architecture of a software system defines that system in terms of computational components and interactions among those components." According to Shaw and Garlan [1996] "an architectural style

defines: a family of systems in terms of a pattern of structural organization; a vocabulary of components and connectors, with constraints on how they can be combined."

Bass, Clements, and Kazman [2003, page 27] define software architecture from a structural point of view:

Definition 2.1.2. ***(Software architecture II)***
"The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

Perry and Wolf [1992] identify three types of architectural building blocks: processing elements, data elements, and connecting elements. Such a structural view of software architecture is opposite to a descriptive view on software architecture. In a descriptive view, the main entities of a software architecture are (principal) design decisions. Taylor, Medvidovic, and Dashofy provide a corresponding definition in [Taylor et al., 2010, page 58]:

Definition 2.1.3. ***(Software architecture III)***
"A software system's architecture is the set of principal design decisions about that system."

Combining the previous definitions, some inherent properties of software architecture can be derived from these definitions. Software architecture:

- is conceptual,
- is about fundamental things or its abstractions,
- exists in some context,
- is high-level design,
- is overall structure of the system,
- is "the structure of the system, including the principles and guidelines governing their design and evolution over time" [Garlan and Perry, 1995],
- is about components and connectors,
- is about interactions (behavior),
- is about the relationship of its elements.

We will later recall the properties in order to synthesize meaningful software architectures.

2.2 Describing Software Architectures

Software architectures have to be documented, communicated, and managed. High-level design and describing principal design decisions [Taylor et al., 2010, page 58] must be captured using a qualified language. An *architecture description language* (ADL) is any means of expression used to describe a software architecture (ISO/IEC/IEEE 42010 [ISO/IEC/IEEE, 2011]). ADLs can be general- or special-purpose. General-purpose languages used for software architectures are the *unified modeling language* (UML) 2.0 [Object Management Group (OMG), 2005] (see for [Clements et al., 2011, pages 431ff]) and the *systems modeling language* (SysML) [Object Management Group (OMG), 2012] (see for [Clements et al., 2011, pages 465ff]).

Medvidovic and Taylor [2000] present a specialized framework for evaluating ADLs especially for component and connector architectures (and C2-architectures)[1] that contains classification and comparison methods. The authors evaluated nine different ADLs and they concluded [Medvidovic and Taylor, 2000, pages 50ff] that no single ADL completely fulfills the identified needs that the authors gathered. We will now discuss some of these and some additional ADLs in more detail but refer to Medvidovic and Taylor [2000] for a complete discussion.

An early survey on ADLs has been presented in [Clements, 1996] and has been further developed in [Bass et al., 2003]. A recent survey is given in [Malavolta et al., 2013].

- The *architectural analysis and design language* (AADL) has been developed as a SAE International standard [Clements et al., 2011, pages 473ff]. It can be used to describe software as well as hardware architectures.

- *Wright* was developed by Allen at the Carnegie Mellon University [Allen, 1997] and supports components, connectors, roles, and ports. Wright uses *communicating sequential processes* (CSP) to specify and formalize the behavior of ports of components.

- Stanford's well-known *Rapide* [Luckham et al., 1995] is an ADL for prototyping of large-scale, distributed concurrent systems.

- *Acme* was developed by Carnegie Mellon University [Garlan et al., 1997] and supports components, connectors and also, in particular, an architectural ontology.

[1] The C2 architecture [Medvidovic et al., 1997, Taylor et al., 2010] is a special component and connector architectural style.

- *xADL* [Dashofy et al., 2002] has been developed by the University of California, Irvine. It uses *extensible markup language* (XML) documents to describe software architectures thus offering interoperability with other ADLs.

- *Darwin* [Magee et al., 1995] developed by a research group at the Imperial College London supports components, interfaces, and bindings. Connectors are not first-class objects within Darwin. The operational semantics is based on π-calculus [Milner, 1980, 1999].

- The idea of using π-calculus for an operational semantic has been further developed by Oquendo [2004] and Oquendo [2008] for π-ADL.

- *PADL* is an ADL developed by Bernardo et al. [2002] that is based on the process algebra EMPA_{gr} which is a mix of two process algebras, CCS by Milner [1989] and CSP by Hoare [1985].

Most of the itemized special-purpose tools (except Darwin by Magee et al. [1995]) support software components and connectors as objects as well as relations among these objects.

A common problem of such special-purpose languages is the classical hen and egg problem that a lack of industrial tools is caused by low dissemination of these special-purpose languages in industry. It can be seen by comparing the total numbers of software tools supporting general-purpose tools like UML 2.0 and of special-purpose tools. Thus, the suitability of UML as an ADL is examined and discussed by Pérez-Martínez and Sierra-Alonso [2004]. That even academia recognizes UML as an ADL can be seen by the fact that UML is listed as an ADL in [Clements et al., 2011]. Also Medvidovic et al. [2002] provide a comprehensive discussion on UML as an ADL on standalone UML as well as on extended UML.

Taking the arguments made into account, we can conclude that a tool for practical industrial applications, should favor UML instead of highly specialized formalisms as an ADL.

2.2.1 Components and Connectors

In the thesis, we regard software architecture as a set of components together with the connectors realizing the interactions among these components similar to Component&Connector (C&C)-Architectures discussed by Taylor et al. [2010]. Hence, we recapitulate the definitions of components and connectors from [Taylor et al., 2010]:

Definition 2.2.1. *(**Software component**) (see for [Taylor et al., 2010, page 69])*
"A software component *is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context."*

We primarily concentrate on characteristics (2) and (3) since our goal is to synthesize connectors for which interfaces are the access point to a component. From here on, we explicitly exclude the component's behavior as protocols in the thesis. But we are going to specify behavior abstract as interface semantics using concepts. Thus, from a connector's point-of-view a component simply consists of a set of provided and required interfaces. Provided interfaces of a component are exposing the components functionality via these interfaces. Whereas its required interfaces are used by the component to receive a functionality provided by other components.

Definition 2.2.2. *(**Software connector**) (see for [Taylor et al., 2010, page 70])*
"A software connector *is an architectural element tasked with effecting and regulating interaction among components."*

A connector must connect provided interfaces of (possibly several) components with suitable required interfaces of some other components, where some of the required or provided interfaces may be optional depending on the usage context.

Definition 2.2.3. *(**Interface type**) An interface with name I has the interface type I.*

Note, that an interface's type is independent of its methods. The reason for this definition is that software architects use the name of an interface, like `IIterator`, as a semantical description of the functionality a component provides via this interfaces.

The following special case, depicted for component C in Figure 2.1, occurs often:

Definition 2.2.4. *(I-**connector**) A connector that connects a single provided interface I to a single required interface I is called an* I-connector.

Example 2.1. *An example I-connector is depicted in Figure 2.1. The connector C requires an interface I and provides an interface I as well.*

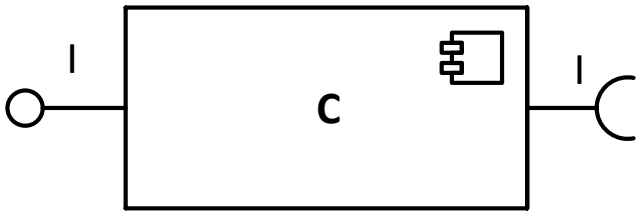


Figure 2.1: I-connector

An I-connector is not necessarily a simple pipe since it may provide further functionality beyond the fact that it connects I to I. For example, it may add security properties, for example a secure channel by using encryption, to the interaction.

Interface types are not sufficient to capture all relevant properties, thus it may be necessary to also include semantic concepts to specify them. Note that the distinction between components and connectors does not have to be sharp. Rather, components and connectors can be regarded as two ends of a continuum as argued by Kell [2007].

Usually, connectors are generic with regard to their usage scenario, i.e., there is a limited number of functional and non-functional *general* properties that a connector may or may not need to have. The idea of composite connectors (connectors that are assembled out of components) has been advocated by Spitznagel and Garlan [2001] and by Julien and Perry [2008], because of such a genericity and in order to improve reusability. Clearly, a high degree of automation in finding appropriate composite connectors would be an advantage. The idea underlying our approach to connector synthesis is to assume a set of building blocks from which composite connectors may be synthesized for a given usage context. Some of the building blocks may be specialized for exactly such an usage context, whereas others may be more generic. The building blocks may be atomic, i.e., they provide certain interfaces without further requirements. Other more complex building blocks may offer certain functionalities if they are provided with certain functionalities offered by other building blocks. Such complex blocks can be used to build compositions of building blocks and may capture architectural design decisions or even whole architectural styles with regard to software connectors. Note that some building blocks can be components or connectors themselves. Since, in principle, building blocks can be composed in arbitrary way the resulting composite connectors may offer great variability. This variability often makes the manual retrieval and composition of building blocks tedious as well as repetitious and therefore error-prone.

We address the problem by exploiting taxonomies to specify properties

of connectors as described by Hirsch et al. [1999] and by Mehta et al. [2000]. The repository has to be thus designed or specified *once* for possibly multiple uses in a given usage context. Each building block is linked to an *underlying* (i.e. hidden from the user by an abstract link) set of code templates, that have been prepared such that they correctly implement the specification of the building block. In the following subsection we will present a rich semantic based description language that can be used to specify building blocks. Once a repository of building blocks has been specified using intersection types we synthesize connectors by asking suitable inhabitation questions. An inhabitant can be regarded as a description or blueprint of how certain building blocks should be composed. From such compositions we infer connector descriptions in an ADL. The ADL-description of a connector is used to generate executable code by combining the underlying templates according to a suitable interpretation of the ADL-description.

In order to avoid confusion, we distinguish the terms specification, synthesis, and generation.

Definition 2.2.5. *(**Specification**)*
We denote a process that constructs a concise logical model including semantics for synthesis from a problem space in a C&C architecture, as specification.

Definition 2.2.6. *(**Synthesis**)*
We denote a process that constructs a composition plan by means of a logical method, as synthesis.

It means that type inhabitation is a synthesis method producing an inhabitant as composition plan.

Definition 2.2.7. *(**Generation**)*
We denote a process that transforms a composition plan into an abstract or textual object, as generation.

The definition also covers the generation of source code, deployment code, and also UML diagrams as concrete generation products.

Summarizing, the Combinatory Logic Connector Synthesis method consists of three essential steps:

1. Using a taxonomy, we semantically *specify* the building blocks in our repository and link them to a set of templates.

2. We *synthesize* compositions of building blocks by posing suitable inhabitation questions.

3. We interpret the resulting inhabitants in an ADL and/or *generate* code from underlying templates.

An overview of the method is depicted as an abstracted process in Figure 2.2.

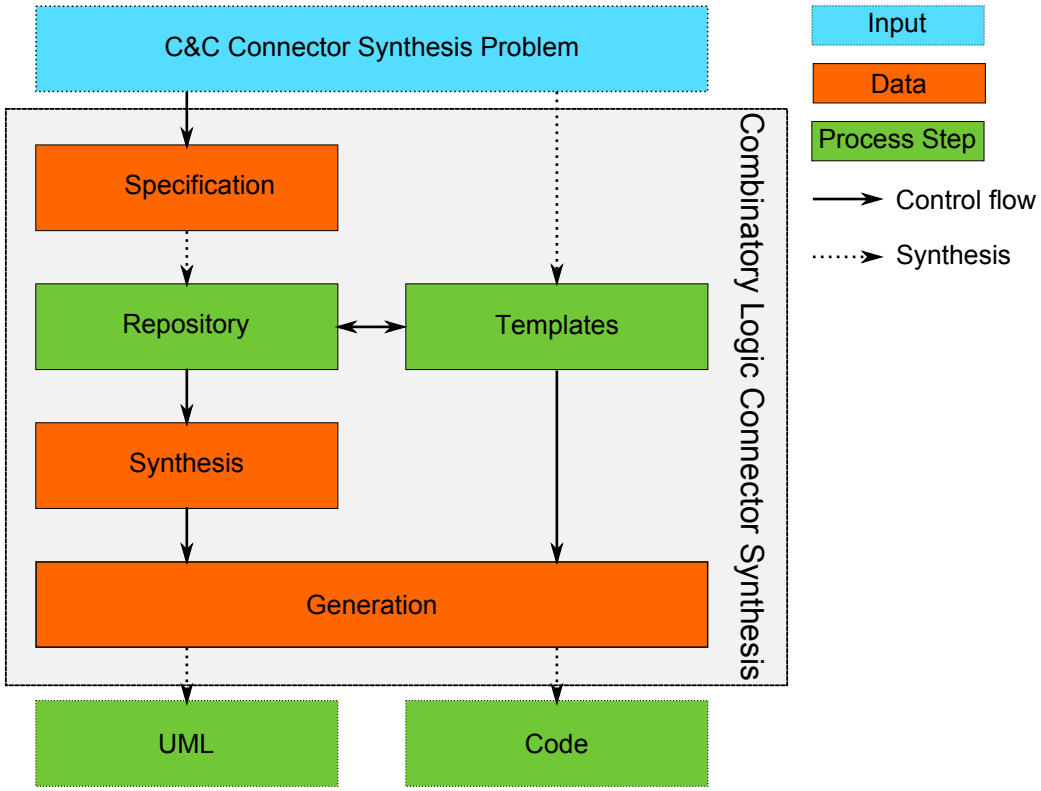


Figure 2.2: Overview of the Combinatory Logic Connector Synthesis method.

We follow the discussion of Taylor et al. [2010] and extend their argumentation for software connectors as first-class citizenship in software architecture.

Software Connectors as First-Class Citizens

A software connector is an architectural element modeling different properties in a software architecture. In Definition 2.2.2 and in [Taylor et al., 2010], a software connector is responsible for the interactions among software components. The interaction is performed by transferring control and/or data among components. The connector also encodes and enforces rules that govern those interactions. Some simple interactions are procedure calls and shared variable access. Some more complex and semantically rich interactions are client-server protocols, database access protocols, and asynchronous event

multicast. Each connector provides interaction via so called ducts. Ducts [Mehta et al., 2000, Kell, 2007] are channels along which data and control can be passed between components.

The reason for distinguishing between connectors and components is that components provide application-*specific* functionality whereas connectors provide application-*independent* interaction mechanisms. The interactions can be more abstract by applying parameterization to connectors. The parametrization allows the specification of complex interactions, because these interactions can be specified independently from the components.

A classification can be used that distinguishes binary from n-ary connectors and asymmetric from symmetric connectors. These concepts will be used later within an ontology of connectors. Different interaction protocols can be identified and are presented in [Taylor et al., 2010]. The global view on an architecture's interaction model is broken down to a set of local interaction definitions. The global architecture's interaction emerges from such local architecture interactions. Then, such ideas can be used for a component system for interactions and their information. Such extra-components are called connectors and in a formal model, connector building blocks. Furthermore, the introduction of connectors brings component independence and interaction flexibility.

The benefits of first-class connectors are the conceptual separation of computation from interaction. It minimizes component interdependencies and supports the evolution of software architectures at component-, connector-, and system-level. It offers potential for supporting dynamism in software architectures and facilitates heterogeneity. Software connectors can become partition points as presented in the examples in Chapter 8. The separation of components and connectors also aids the system's analysis and testing.

Connectors allow modeling of arbitrarily complex interactions because of a local view on interactions. The connector's flexibility, for example by composition or by individual local adaptations, aids the system's evolution. Components can be added, removed, replaced, reconnected, and migrated without affecting other components, because connectors isolate effects. A support for connector interchange is also desirable. Then, connectors could be interchanged without affecting the system's functionality. This can be advantageous with respect to the system's evolution and also its prototyping.

Also Kell [2007] advocates software connectors as first-class citizens in software architecture and provides a table of connector types and their instances. Balek and Plasil [2001] discuss the importance and effects of software connectors for the deployment of the software. A more general discussion on software connectors with various examples can be found in [Balek, 2002].

2.2.2 Software Connector Roles

With a first-class citizenship, software connectors are now the locus of interaction among a set of components. The behavior and interaction of components is facilitated by connectors and their sometimes implicit protocol specification that defines the connector's properties. Connectors can be characterized by the different types of interfaces they are able to mediate. A connector has to assure its interaction policy for a desired system's functionality. Therefore, rules about interaction ordering and interaction commitments, for instance, about performance or on availability are essential. And last but not least, software connectors play different roles in software architectures. According to Taylor et al. [2010], the software connector roles can be identified as

- Communication
- Coordination
- Conversion
- Facilitation.

The order of the roles does not correspond to the importance of a role. Their importance varies with different applications. The roles are orthogonal to each other and will now be discussed in more detail.

Connectors as Communicators

The most prominent role associated with connectors is the support of different communication mechanisms, for example *procedure call* (PC), *remote procedure call* (RPC), shared data access, and message passing. Software connectors pose constraints on communication structures and directions, for instance pipes. They put constraints on the quality of service, for instance persistence, availability, and reliability. The concept separates communication of systems from their computation but may as well influence the non-functional system characteristics, by way of example performance, scalability, and security.

Connectors as Coordinators

Software connectors also take the role of coordinators. Connectors coordinate components by determining computational control. Furthermore, connectors also control the delivery of data among components and are the architectural element that separates control from computation. The coordination role itself is orthogonal to the other roles, communication, conversion, and facilitation. Whereas the elements of control reside in the three remaining roles: communication, conversion, and facilitation.

Connectors as Converters

Software connectors also serve as converters among components. Connectors enable interaction of independently developed and possibly mismatched components. The mismatches are caused by the component's interactions. The conversion performed by a connector can be versatile. Possible conversions are the conversion of types, of numbers, of the frequency, or of the order of interactions. A number of different converters are known in software architecture, for example adaptors or wrappers. For example, Taylor et al. [2010] and Buschmann et al. [1996] describe various converters primarily as architectural patterns. We will later use the term adapter and mean in general a converter.

Connectors as Facilitators

The last role in the list of roles is the connector as a facilitator. Connectors enable interactions among components that are intended to interoperate. Therefore, connectors have to mediate and to streamline the interactions and also to govern access to shared information. With respect to non-functional requirements, connectors also have to ensure proper performance profiles, for instance, load balancing. Connectors have to provide synchronization mechanisms like critical sections, transactions, and monitors, because software architectures include such concurrent components.

The different roles are used to define dimensions and sub-dimensions in software connector taxonomies. The usefulness of software connector taxonomies, in particular, for synthesizing software connectors will be discussed next.

2.2.3 Software Interconnection Models

Software *interconnection models* (IMs) are relevant for the understanding of software connectors. The interconnections in a software system can be classified according to Perry [1987]. Interconnection can be found on different abstraction levels and on disparate architectural views on a software architecture. Perry discriminates between three essential classes of interconnections. The classes are ordered by the contained information content.

Unit interconnection

A unit interconnection defines relations between different units of the system. In such a model, units are components that are modules or files and a basic unit relationship is the dependency of these units. The model is programming language agnostic and does not contain semantic information.

Syntactic interconnection

A syntactic interconnection describes relations among syntactic elements of programming languages, for example, variable definition and use, or method declaration and invocation.

Semantic interconnection

A semantic interconnection expresses how the system's components are meant to be used. This includes the view of the component's designer, his intentions, prescriptive view, and captures how system components are actually used, descriptive view. In addition, it also includes the intention of a component's user. Such interconnection semantics can be formally specified by pre- and post-conditions or dynamic interaction protocols, for example *communicating sequential processes* (CSP), *finite state machine* (FSM), or UML state diagrams. The semantic connections build on the syntactic interconnections. Furthermore, interconnections can be static and/or dynamic.

For a meaningful and intended software system, a complete interconnection specification is needed that specifies both syntactic and semantic interconnection validity. Such a specification is crucial at every level of software architectures and allows the construction of large components with complex interactions. It facilitates dealing with heterogeneity in software architecture and increasing the reusability of components.

In result, a formal semantic model including concepts arises as a requirement for using semantic interconnections. Ontologies and taxonomic hierarchies are such formal semantic models and will be discussed in Section 2.4 on the facing page. Specialized taxonomies will play an important role as ontological structure in the presented method.

2.2.4 Composing Basic Connectors

The composition of systems eases the system designer's work. The number of involved elements can be reduced by composing composite components from a reduced set of atomic components. Mehta and Medvidovic [2003] present a method for composing architectural styles from architectural primitives. This work is based on an ADL named *Alfa*.

In many systems, a software connector incorporating multiple types may be required to service (a subset of) the components. The compositional approach is not a universal remedy, because it cannot be expected that all connectors can be composed and some might be naturally interoperable or even incompatible. Thoughtful design and trade-offs are required. Composition can be considered

at the level of connectors in all type dimensions and subdimensions. For example, a composition for component-based modeling is presented Gößler and Sifakis [2005] which is focused on refinement and on interaction models. In addition, Sifakis [2005] provides a related framework for component-based construction based on an algebraic composition model for interaction models.

From an orthogonal viewpoint of software performance engineering, the authors Strittmatter and Happe [2012] discuss a compositional approach to software connectors, because the choice of software connectors has as well a major impact on the system's performance.

2.3 Synthesizing Software Architectures

Synthesis combines one or more entities to form something new. In order to automate synthesis some prerequisites have to be fulfilled. Knowledge about the domain and the context of the entities and the entities itself must be represented. A synthesis goal must be specifiable. The synthesis (-process) needs an operational semantic. An automatic synthesis procedure must use the represented knowledge and the target to create a combination of components coming from a set of components. This combination must be interpreted in the domain and the context of the entities.

In order to synthesize meaningful software architectures, synthesis must reflect the properties listed in Section 2.1 on page 19. It has to operate on abstract, conceptual objects and to support contexts. The structural development and high-level design with components and connectors as well as their mutual relationships must also be supported.

The usage of ontologies for capturing knowledge in software architecture is not new. The ADL Acme by Garlan et al. [1997] supports an architectural ontology and the usage of ontologies to capture architectural design decisions has been proposed by Kruchten [2004]. An idea for the composition of architectural aspects based on style semantics is presented by Chavez et al. [2009].

ADLs are ideal candidates as interpretation targets of compositions of software architectural entities because ADLs come with a defined semantic and a restrictable meaning.

2.4 Ontologies in Software Architecture

In the subsection introducing semantic interconnections, a requirement for complete interconnection specification is raised also including the semantics of

the connectors.The use of an ontology for the architectural views is presented in [Kruchten, 2004].

Taxonomic hierarchies of classes, among others, are often equated with ontologies according to Gruber [1993]. In the thesis (and further referenced works using ontologies), we equate taxonomy or taxonomic hierarchy with ontology and depict taxonomies as taxonomic trees.

A specific taxonomy for connectors have been introduced by Mehta et al. [2000] based on a previous classification model provided by Hirsch et al. [1999]. The book of Taylor et al. [2010] comprises a complex taxonomical model in the form of taxonomic trees. The connector taxonomies have in common that the knowledge of a software architect is captured and can be algorithmically used for reasoning.

Inverardi and Tivoli [2013] also use an ontology for the automated synthesis of modular connectors. However, the connectors differ in their abstraction level from the architectural connectors presented in our work. Spalazzese and Inverardi [2010] and Inverardi and Tivoli [2013] consider compositions of mediators (adaptors) on method level focusing on interaction and interactions based on an observable behavior of traces.

The taxonomy in [Taylor et al., 2010] and a derived, specialized taxonomy that is used for Combinatory Logic Connector Synthesis are discussed in Section 6.2.5 on page 126. This taxonomy will include conceptual information on functional and non-functional properties of a connector as well as its technological concepts.

Chapter 3

Bounded Combinatory Logic

This chapter establishes the mathematical foundation for combinatory logic synthesis. We begin with the introduction of intersection types that is the foundation of bounded combinatory logic. A decision problem for this logic called relativized type inhabitation and a decision procedure for it will be discussed. Afterwards, the connection between relativized type inhabitation and synthesis will be reviewed.

3.1 Combinatory Logic

Before we explain the mathematical foundation, we refer to the discussed motivation of combinatory logic in the introduction in Section 1.1 on page 3 and combinatory logic synthesis in Section 1.2 on page 5. We now briefly recapitulate the logical and type-theoretical arguments of the discussion made in these two sections for a motivation of the usage of combinatory logic for synthesis.

In this work we pursue the way of a type-based approach to deductive logic synthesis that is used for synthesizing compositions of typed components employing intersection types. These components in a repository shall be specified by semantic concepts coming from a taxonomy. The usage of a logic for the specification of components and deduction for the synthesis suggests itself.

Usually a fixed set of basic combinators or axiom schemes like **SK** in Subsection 1.1 are considered in combinatory logic. Under the Curry-Howard isomorphism (propositions-as-types correspondence), the provability of formulas in this logic is PSPACE-complete for simple types as shown by Statman [1979]. The inhabitation problem for the λ-calculus with intersection types was shown to be undecidable by Urzyczyn [1999]. Combinatory logic with

System	Complexity	Author(s)
$\mathrm{FCL}(\leq)$	PTIME	Rehof and Urzyczyn [2011a]
$\mathsf{FCL}(\cap, \leq)$	EXPTIME	Rehof and Urzyczyn [2011a]
$\mathrm{BCL}_k(\leq)$	EXPTIME	Düdder et al. [2012]
$\mathsf{BCL}_k(\cap, \leq)$	(k+2)-EXPTIME	Düdder et al. [2012]
$\mathrm{CL}(\mathbf{SK}) \rightarrow$	PSPACE	Statman [1979]
$\lambda(- \cap I)$	EXPSPACE	Rehof and Urzyczyn [2012]
$\lambda^{r2}\cap$	EXPSPACE	Urzyczyn [2009]
$\lambda\cap$	∞	Urzyczyn [1999]
$\mathrm{CL}(\cap)$	∞	Dezani-Ciancaglini and Hindley [1992] and Urzyczyn [1999]

Table 3.1: Complexity results for various provability (inhabitation) problems

intersections types was studied by Dezani-Ciancaglini and Hindley [1992], where it was shown that the combinatory system is complete in that it is logically equivalent to the λ-calculus by term (proof) translations in both directions. It follows that provability (inhabitation) for combinatory logic with intersections types is undecidable. If arbitrary sets of axiom schemes are considered, then the inhabitation problem is undecidable even for simple types ($\lambda_{\rightarrow}$) as described in Section 1.1 on page 3 shown by Linial and Post [1949] (Linial-Post theorem). Table 3.1 lists some complexity results on deciding provability (inhabitation) in various systems.

For composition, a logic with modus ponens is meaningful, because we can interpret the usage of modus ponens as applying a composition operation.

Under the Curry-Howard isomorphism, combinator types correspond to axiom schemes of propositional logic in a Hilbert-style proof system, with modus ponens and a rule of axiom scheme instantiation as the principles of deduction. The schematic interpretation of axioms corresponds to implicit polymorphism of combinator types, where type variables ($\alpha, \beta, \gamma, \ldots$) may be instantiated with arbitrary types. Thus, the combinator $\mathbf{K}$ has types $\tau \rightarrow \sigma \rightarrow \tau$ for all τ and σ. In logic, implicit polymorphism is termed typical ambiguity.

Restricting combinatory logic with intersection types to a monomorphic system forming a Hilbert-style logic has been studied by Rehof and Urzyczyn [2011a]. The resulting logic with intersection types and subtyping is called finite combinatory logic $\mathsf{FCL}(\cap, \leq)$. Its provability problem is decidable in EXPTIME.

Bounded combinatory logic $\mathsf{BCL}_k(\cap, \leq)$ extends $\mathsf{FCL}(\cap, \leq)$ by a restricted

form of polymorphism but avoids undecidability. Bounded combinatory logic restricts substitutions of variables by a bound on the depth of types that are instantiated by substitutions. Düdder et al. [2012] showed that synthesis in $\mathsf{BCL}_k(\cap, \leq)$ is $(k+2)$-EXPTIME-complete.

A way to refine the specification of components is presented in the following section. Whereas in addition, in the next chapter different heuristical optimization approaches for the synthesis problem are presented, that allow the solution of the synthesis problem under a reasonable resource consumption in practical applications. To understand these optimizations and for discussing the theoretical problems, this chapter contains the mathematical foundation and begins with the basics of the combinatory logic and its definitions.

3.2 Intersection Types

The current section is mainly reproduced from our paper that introduces $\mathsf{BCL}_k(\cap, \leq)$ [Düdder et al., 2012], but is expanded with more details, for example detailed proofs, and examples. Further details on $\mathsf{BCL}_k(\cap, \leq)$ can be found in technical report [Düdder et al., 2012].

Intersection types have been introduced by Barendregt, Coppo, and Dezani-Ciancaglini [1983] to define a type system which characterizes exactly strong normalizing λ-terms. The system types the strongly normalizing terms [Coppo and Dezani-Ciancaglini, 1980, Pottinger, 1980], hence typability is undecidable. In contrast, typability in simple types, $\lambda_{\rightarrow}$, is decidable in linear time, so there is an immense gap between simple types and intersection types from the perspective of typability.

The connection between strong normalization and program termination in λ-calculus with typeability in intersection types provides an indication on the expressibility of intersection types with regard to specification of computations.

The inhabitation problem for λ-calculus with intersection types is closely related to the λ-definability problem [Salvati, 2009, Salvati et al., 2012] and is also undecidable [Urzyczyn, 1999]. In contrast, deciding the inhabitation problem for simple types is PSPACE-complete [Statman, 1979].

Intersection types are regarded interesting for the overarching synthesis purposes presented in this thesis, because intersection types are known to capture deep semantic properties of λ-terms [Dezani-Ciancaglini and Hindley, 1992].

3.2.1 Types

Type expressions, ranged over by τ, σ etc., are defined by

$$\tau, \tau' \ ::= \ a \mid \omega \mid \tau \to \tau' \mid \tau \cap \tau' \mid C(\cdots)$$

where $a, b, c, \ldots$ range over *atoms* comprising of *type constants*, drawn from a finite set $\mathbb{A}$ including the constant ω, and *type variables*, drawn from a disjoint denumerable set $\mathbb{V}$ ranged over by $\alpha, \beta, \gamma, \ldots$, and type constructors drawn from a disjoint denumerable set $\mathbb{C}$ ranged over $C, C', \ldots$. We let $\mathbb{T}$ denote the set of all types.

As usual [Barendregt et al., 1983], types are taken modulo commutativity ($\tau \cap \sigma = \sigma \cap \tau$), associativity ($(\tau \cap \sigma) \cap \rho = \tau \cap (\sigma \cap \rho)$), and idempotentcy ($\tau \cap \tau = \tau$). As a matter of notational convention, function types associate to the right, and $\cap$ binds stronger than $\to$.

Let $Var(\tau)$ and $At(\tau)$ denote, respectively, the set of variables and the set of atoms occurring in τ.

A type $\tau \cap \sigma$ is said to have τ and σ as *components*. For an intersection of several components we sometimes write $\bigcap_{i=1}^{n} \tau_i$ or $\bigcap_{i \in I} \tau_i$ or $\bigcap\{\tau_i \mid i \in I\}$, where the empty intersection ($\bigcap_{i \in \emptyset}$) is identified with ω.

3.2.2 Subtyping

Intersection types come with a natural notion of *subtyping* $\leq$ as given by Barendregt, Coppo, and Dezani-Ciancaglini [1983] and presented by Barendregt et al. [2013]. Subtyping $\leq$ is the least preorder (that is a reflexive and transitive relation) on $\mathbb{T}$, satisfying the following axioms with $\tau, \tau', \sigma, \sigma', \rho, \omega \in \mathbb{T}$:

$$\sigma \leq \omega, \quad \omega \leq \omega \to \omega, \quad \sigma \cap \tau \leq \sigma, \quad \sigma \cap \tau \leq \tau, \quad \sigma \leq \sigma \cap \sigma;$$

$$(\sigma \to \tau) \cap (\sigma \to \rho) \leq \sigma \to \tau \cap \rho;$$

$$\text{If } \sigma \leq \sigma' \text{ and } \tau \leq \tau' \text{ then } \sigma \cap \tau \leq \sigma' \cap \tau' \text{ and } \sigma' \to \tau \leq \sigma \to \tau'.$$

Type constant ω is the least-upper bound with respect to the subtyping relation $\leq$. We may extend the subtyping relation $\leq$ on $\mathbb{T}$ by $a \leq a'$ with $a, a' \in \mathbb{A}$ by adding the axiom $(a, a') \in \mathbb{A} \Rightarrow a \leq a'$ to the axioms given in Barendregt et al. [1983]. Subtyping is shown decidable in Ptime by Rehof and Urzyczyn [2011a].

We identify σ and τ, written as $\sigma = \tau$, when $\sigma \leq \tau$ and $\tau \leq \sigma$. The relation $=$ is referred to as *type equality*. We write $\sigma \equiv \tau$, if σ and τ are *syntactically* identical. For $\tau \leq \sigma$ we will say:

- that τ is a *subtype* of σ and vice versa
- that σ is a *supertype* of τ.

The following two convenient distributivity properties follow from the previous axioms of subtyping:

$$(\sigma \to \tau) \cap (\sigma \to \rho) = \sigma \to (\tau \cap \rho)$$

$$(\sigma \to \tau) \cap (\sigma' \to \tau') \leq (\sigma \cap \sigma') \to (\tau \cap \tau')$$

We extend the subtyping relation $\leq$ with additional covariant type constructors $C(\cdots) \in \mathbb{C}$. These type constructors have arity $n \geq 0$ but do not distribute over $\to$ and $\cap$. We identify type constructors with 0-arity with type constants.

We say that a type τ is *reduced with respect to* ω if it has no subterm of the form $\rho \cap \omega$ or $\tau_1 \to \cdots \to \tau_m \to \omega$ with $m \geq 1$. It is easy to reduce a type with respect to ω, by applying the equations $\rho \cap \omega = \rho$ and $\tau_1 \to \cdots \to \tau_m \to \omega = \omega$ left to right.

We augment the subtyping relation $\leq$ from [Barendregt et al., 1983] by covariant type constructors.

Definition 3.2.1. *(**Subytping relation** $\leq_C$)*
The subtyping relation $\leq_C$ is the smallest transitive closure of the subtyping relation $\leq$ with covariant type constructors $C \in \mathbb{C}$ with n the number of arguments of C and the additional subtyping rule:

$$\tau_1 \leq_C \sigma_1, \ldots, \tau_n \leq_C \sigma_n \Rightarrow C(\tau_1, \ldots, \tau_n) \leq_C C(\sigma_1, \ldots, \sigma_n)$$

We have to show that the extension from $\leq$ to $\leq_C$ does not change the complexity of deciding the subtype relation.

Lemma 3.2.1. *The subtyping relation $\leq_C$ for intersection types is decidable in* PTIME.

Proof. The first observation is that for a covariant type constructor linearly (in the number of the arguments) many checks of the subtyping relation $\leq_C$ must be performed, because these covariant type constructors are not distributive. Even, in nested covariant type constructors can occur linearly many times. Therefore, at most linear many arguments of covariant type constructors exist in a type. Assume type τ has k and type σ has l arguments. Then at most $k \times l$ subtype checks $\leq$ must be performed. The subtype $\leq$ check is computable in PTIME [Rehof and Urzyczyn, 2011a]. Therefore, deciding $\leq_C$ is also in PTIME. □

The following property, probably first stated in [Barendregt et al., 1983], is often called *beta-soundness*. Note that the converse is trivially true.

Lemma 3.2.2. *Let a and a_j, for $j \in J$, be atoms.*

1. *If $\bigcap_{i\in I}(\sigma_i \to \tau_i) \cap \bigcap_{j\in J} a_j \leq a$ then $a = a_j$, for some $j \in J$.*
2. *If $\bigcap_{i\in I}(\sigma_i \to \tau_i) \cap \bigcap_{j\in J} a_j \leq \sigma \to \tau$, where $\sigma \to \tau \neq \omega$, then the set $\{i \in I \mid \sigma \leq \sigma_i\}$ is nonempty and $\bigcap\{\tau_i \mid \sigma \leq \sigma_i\} \leq \tau$.*

3.2.3 Paths

If $\tau = \tau_1 \to \cdots \to \tau_m \to \sigma$, then we write $\sigma = tgt_m(\tau)$ and $\tau_i = arg_i(\tau)$, for $i \leq m$. We say that σ is a *target* of τ and τ_i is the *i-th argument* of τ.

If $arg_i(\tau) = \rho$ for all i we also write $\tau = \rho^m \to \sigma$. A type of the form $\tau_1 \to \cdots \to \tau_m \to a$, where $a \neq \omega$ is an atom,[1] is called a *path of length m*. A type τ is *organized* if it is a (possibly empty) intersection of paths (those are called *paths in τ*).

Lemma 3.2.3. *Every type τ is equal to an organized type $\overline{\tau}$, computable in polynomial time [Rehof and Urzyczyn, 2012].*

Proof. Define $\overline{a} = a$ if a is an atom and let $\overline{\tau \cap \sigma} = \overline{\tau} \cap \overline{\sigma}$. If $\overline{\sigma} = \bigcap_{i\in I} \sigma_i$ then take $\overline{\tau \to \sigma} = \bigcap_{i\in I}(\tau \to \sigma_i)$. □

Note that premises in an organized type do not have to be organized, i.e., organized is not necessarily the same as normalized in the sense of Hindley [1982]. Essentially, $\overline{\tau}$ is obtained by repeatedly applying the distributivity property above to any intersection occurring as a target of τ.

For an organized type σ, we let $\mathbb{P}_m(\sigma)$ denote the *set of all paths* in σ of length m or more. We extend the definition to arbitrary τ by implicitly organizing τ, i.e., we write $\mathbb{P}_m(\tau)$ as a shorthand for $\mathbb{P}_m(\overline{\tau})$.

The *size* of a type τ, denoted $|\tau|$, is defined to be the number of nodes in the syntax tree of τ (this is identical to the textual size of τ). The *path length* of a type τ is denoted $\|\tau\|$ and is defined to be the maximal length of a path in τ.

[1] Note that $\tau_1 \to \cdots \to \tau_m \to \omega = \omega$.

3.2.4 Substitutions

A *substitution* as given in [Düdder et al., 2012] is a function $S : \mathbb{V} \to \mathbb{T}$ such that S is the identity everywhere but on a finite subset of $\mathbb{V}$. For a substitution S, we define the *support* of S, written $Supp(S)$, as $Supp(S) = \{\alpha \in \mathbb{V} \mid \alpha \neq S(\alpha)\}$. We may write $S : V \to \mathbb{T}$ when V is a finite subset of $\mathbb{V}$ with $Supp(S) \subseteq V$. With $At(\tau)$ defining the set of type atoms occurring in τ we write $At(S)$ to denote the set $\{At(S(\alpha)) \mid \alpha \in Supp(S)\}$. A substitution S is tacitly lifted to a function on types, $S : \mathbb{T} \to \mathbb{T}$, by homomorphic extension.

Definition 3.2.2. *(**Application of substitution**)*
An application of a substitution $S = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ on types to a type σ is defined as follows:

$$
\begin{aligned}
&S(a) && = a \text{ if } a \in \mathbb{A}\\
&S(\alpha_i) && = \tau_i \text{ if } \alpha_i \in \mathbb{V}\\
&S(\bigcap_{j\in J} \sigma_i) && = \bigcap_{j\in J} S(\sigma_j)\\
&S(\sigma_1 \to \sigma_2) && = S(\sigma_1) \to S(\sigma_2)\\
&S(C(\sigma_1, \ldots, \sigma_m)) && = C(S(\sigma_1), \ldots, S(\sigma_m)) \text{ if } C \in \mathbb{C}.
\end{aligned}
$$

A type σ' with $\sigma' = S(\sigma)$ is called an *instantiation* of a type σ under substitution S.

A *kinded type variable* α of *kind* κ_A with a set of type atoms $A \subseteq \mathbb{A}$, written as $\alpha.\kappa_A$, restricts the set of type atoms for substitutions applied on α to A. We may omit A in $\alpha.\kappa_A$ and write $\alpha.\kappa$ if A is clear from the context of κ. With $\mathbb{T}_A$ we denote all intersection types that can be constructed with $A \subseteq \mathbb{A}$ ($\mathbb{T}_A = \{t \in \mathbb{T} \mid At(t) \subseteq A\}$). The following definition extends Definition 3.2.2 to kinded variables as follows:

Definition 3.2.3. *(**Substitution of kinded variables**)*
A substitution S for a kinded type variable $\alpha.\kappa_A$ of kind κ_A satisfies

$$S(\alpha.\kappa_A) \in \mathbb{T}_A.$$

We canonically extend substitutions to sets of kinded type variables using Definition 3.2.3 compatibly. For convenience, we use anonymous substitutions S and write $\{\alpha \mapsto S(\alpha) \mid \alpha \in \mathbb{V}\}$.

3.3 Bounded Combinatory Logic

After introducing intersection types, we can define bounded combinatory logic. As motivated in Section 1.2 on page 5, term application represents

composition and typed combinators represent components in a repository of semantically annotated components. The current section will make this representation more precise.

Applicative terms ranged over by $e, e', \ldots$ are defined by

$$e \ ::= \ x \mid (e\ e).$$

Here $x, y, z, \ldots$ range over a denumerable set of *combinators* and $(e\ e)$ is referred to as an *application*. We use the convention that application associates to the left and we freely omit outermost parentheses. Thus, any applicative term can be written uniquely as $x\ e_1 \ldots e_n$. We freely denote $x\ e_1 \ldots e_n$ by $x(e_1, \ldots, e_n)$, representing x as an n-ary function. A *type environment* Γ is a finite set of *type assumptions* of the form $x : \tau$ interpreted as x has type τ.

Definition 3.3.1. ***(Type levels)***
Given a type τ, we define the level *of τ, written $\ell(\tau)$, as follows.*

$$\begin{array}{lcl} \ell(a) & = & 0, \text{ for } a \in \mathbb{A} \cup \mathbb{V}; \\ \ell(\tau \to \sigma) & = & 1 + \max\{\ell(\tau), \ell(\sigma)\}; \\ \ell(\bigcap_{i=1}^{n} \tau_i) & = & \max\{\ell(\tau_i) \mid i = 1, \ldots, n\}. \end{array}$$

The level of a substitution S, written $\ell(S)$, is defined as

$$\ell(S) = \max\{\ell(S(\alpha)) \mid \alpha \in \mathbb{V}\}.$$

A level-k type *is a type τ with $\ell(\tau) \leq k$, and a* level-k substitution *is a substitution S with $\ell(S) \leq k$. For $k \geq 0$, we let $\mathbb{T}_k$ denote the set of all level-k types. For a subset A of atomic types, we let $\mathbb{T}_k(A)$ denote the set of level-k types with atoms (leaves) in the set A.*

Notice that the level of a type is independent from the width (number of arguments) of intersections. Notice also that $\ell(S)$ is completely determined by the restriction of S to $Supp(S)$: if $Supp(S) = \emptyset$, then $\ell(S) = 0$, and if $Supp(S) \neq \emptyset$, then $\ell(S) = \max\{\ell(S(\alpha)) \mid \alpha \in Supp(S)\}$. Finally, we have $\ell(S \circ S') \leq \ell(S) + \ell(S')$ where $\circ$ denotes function composition.

3.3.1 Type Assignment

For each $k \geq 0$ the system $\mathsf{BCL}_k(\cap, \leq)$ (*k-bounded combinatory logic with intersection types*) is defined by the type assignment rules shown in Figure 3.1. In rule (var), the condition $\ell(S) \leq k$ is understood as a side condition to the axiom $\Gamma, x : \tau \vdash_k x : S(\tau)$. The restriction to *simple types* (types without $\cap$) is called $\mathsf{BCL}_k(\leq, \to)$ and is defined by the rules (var), ($\to$E) and ($\leq$), where

τ and τ' range over simple types, by dropping all axioms from the subtyping relation that involve $\cap$, and by considering only substitutions S mapping type variables to simple types. Recalling from Rehof and Urzyczyn [2011a] *finite combinatory logic with intersection types*, denoted $\mathsf{FCL}(\cap, \leq)$, can be presented as the restriction of $\mathsf{BCL}_k(\cap, \leq)$ in which the (var) rule is simplified to the axiom $\Gamma, x : \tau \vdash_k x : \tau$ in which substitution is disallowed.

Under the type assumptions in Γ, the statement $\Gamma \vdash_k e : \tau$ denotes that the applicative term e can be given the type τ.

$$\frac{[\ell(S) \leq k]}{\Gamma, x : \tau \vdash_k x : S(\tau)}(\text{var}) \qquad \frac{\Gamma \vdash_k e : \tau \to \tau' \quad \Gamma \vdash_k e' : \tau}{\Gamma \vdash_k (e\ e') : \tau'}(\to\text{E})$$

$$\frac{\Gamma \vdash_k e : \tau_1 \quad \Gamma \vdash_k e : \tau_2}{\Gamma \vdash_k e : \tau_1 \cap \tau_2}(\cap\text{I}) \qquad \frac{\Gamma \vdash_k e : \tau \quad \tau \leq \tau'}{\Gamma \vdash_k e : \tau'}(\leq)$$

Figure 3.1: Bounded combinatory logic $\mathsf{BCL}_k(\cap, \leq)$

3.3.2 Relativized Inhabitation Problem

The *inhabitation problem* is the following decision problem:

Assuming a fixed Γ*, given a type* τ *as input, is there a term* e *with* $\Gamma \vdash e : \tau$*?*

The term e is called an *inhabitant (of* τ*)*. Note, that the type environment Γ is not part of the input to the inhabitation problem and contains a fixed set of combinators like **SK**. In contrast, the *relativized inhabitation problem* is the following decision problem abbreviated as $\Gamma \vdash ? : \tau$:

Given Γ *and* τ*, is there a term* e *with* $\Gamma \vdash e : \tau$*?*

The term e is also called an *inhabitant (of* τ*)* and type τ is an *inhabitation goal*, request, or question. In the relativized inhabitation problem, the type environment Γ is also part of the input and makes the decision problem harder to solve with respect to the complexity class of the problem. From here, we will refer to the relativized inhabitation problem whenever we will use the notion *inhabitation* (problem).

The power of using intersection types as type constructs is demonstrated in a brief example. The example contains an excerpt of the tracking object example presented in Section 1.3 on page 7 and is discussed in more detail.

Example 3.1. *Assume a type environment* Γ *with the following definition and concrete, native types* `tempFh` *and* `tempCel` *denoting a temperature in*

degree Fahrenheit respectively in Celsius, `int`*, and* `double`

$$\Gamma = \{ \quad Conv : \texttt{tempFh} \to \texttt{tempCel} \cap \texttt{int} \to \texttt{double}, \quad M : \texttt{tempFh} \cap \texttt{int} \quad \}$$

If we now ask the relativized inhabitation question $\Gamma \vdash ? : double \cap tempCel$*, then a direct inhabitation try will fail, because no combinator in the type environment* Γ *has this goal type. However, by applying the* $(\cap E)$*-rule to both sides of* M *as well as on both sides of Conv and by using the* $(\cap I)$*-rule on the conclusions, we can conclude that* $\Gamma \vdash Conv\, M : \texttt{double} \cap \texttt{tempCel}$ *holds and the inhabitant is* $Conv\, M$*. The specification by intersection types leads to non-trivial and not directly obvious inhabitants.*

Without going into the formal definitions, we present a small example illustrating these ideas and the definitions above for synthesizing software connectors. This example anticipates a more detailed definition of the synthesis method as well as comes back to the discussion of software connector synthesis in the previous chapter.

Example 3.2. *Assume we are given two interfaces* I *and* I' *of two separate components that have to be connected. We are further given a building block* b *that implements such a connection if it is provided with interfaces* I_1 *and* I_2 *and two (software) components* C_1 *respectively* C_2 *that provide these two interfaces. We further assume that* C_1 *is an adapter (a) and* C_2 *is a (transaction) monitor (m), and that* b *preserves the properties of the interfaces it requires. The scenario can be described by the following typed repository* $\Gamma = \{\mathsf{b} : (I_1 \to I_2 \to I' \to I) \cap (\alpha \to \beta \to \alpha \cap \beta), \mathsf{C}_1 : I_1 \cap a, \mathsf{C}_2 : I_2 \cap m\}$*. By asking the inhabitation question* $\Gamma \vdash ? : (I' \to I) \cap a \cap m$ *we ask for a connection between* I' *and* I *that is an adapter and a monitor as well. We obtain the inhabitant* $\mathsf{b}(\mathsf{C}_1, \mathsf{C}_2)$*, describing how the building block and the components have to be composed.*

The following lemma is important for optimization and states that, whenever a type is not inhabited, all its subtypes are also not inhabited.

Lemma 3.3.1. *Let* τ *and* τ' *be types with* $\tau \leq \tau'$*.*
If $\Gamma \not\vdash ? : \tau'$ *holds, then* $\Gamma \not\vdash ? : \tau$ *also holds.*

Proof. Assume there exists an applicative term M with $\Gamma \vdash M : \tau$. Using the $(\leq)$-rule in Figure 3.1, we can conclude $\Gamma \vdash M : \tau'$, which contradicts the initial assumption. Therefore, there cannot exist an applicative term M with $\Gamma \vdash M : \tau$. □

We will also need to refer often to a special case of type inhabitation and therefore introduce the following definition.

Definition 3.3.2. ***(Direct inhabitation)***
An inhabitation question $\Gamma \vdash ? : \tau$ *is* inhabited directly *if* $\Gamma \vdash ? : \tau$ *is inhabited and the inhabitant* e *is a single combinator.*

Proposition 3.3.2. *The relativized inhabitation problem for* $\mathsf{BCL}_k(\cap, \leq)$ *with* $\leq_C$ *subtyping is* $(k+2)-$EXPTIME*-complete.*

Proof. Because $\leq_C$ and $\leq$ are decidable in PTIME, the subtyping relation $\leq$ in the relativized inhabitation problem for $\mathsf{BCL}_k(\cap, \leq)$ can be substituted by $\leq_C$ without changing the complexity class of the problem. □

We do not distinguish both inhabitation problems for $\leq$ and $\leq_C$ and simply write $\leq$.

3.4 Alternating Turing Machines

The synthesis algorithm in the next section uses an *alternating Turing machine* (ATM) as formal computation model. An ATM as proposed in [Chandra et al., 1981] is a tuple $\mathcal{M} = (\Sigma, Q, q_0, q_a, q_r, \Delta)$. The set of states $Q = Q_\exists \uplus Q_\forall$ is partitioned into a set $Q_\exists$ of existential states and a set $Q_\forall$ of universal states.

There is an initial state $q_0 \in Q$, an accepting state $q_a \in Q_\forall$, and a rejecting state $q_r \in Q_\exists$.The *successor* relation $\mathcal{C} \Rightarrow \mathcal{C}'$ on configurations is defined as usual according to the transition relation Δ [Papadimitriou, 1994].

The notion of *eventually accepting* configuration is defined by induction as follows:

- An accepting configuration is eventually accepting.
- If $\mathcal{C}$ is existential and some successor of $\mathcal{C}$ is eventually accepting then so is $\mathcal{C}$.
- If $\mathcal{C}$ is universal and all successors of $\mathcal{C}$ are eventually accepting then so is $\mathcal{C}$.

An input is said to be *accepted* by $\mathcal{M}$ if and only if the initial configuration is eventually accepting.

Formally we define the set of all eventually accepting configurations as the smallest set satisfying the appropriate closure conditions.

3.5 Deciding Relativized Inhabitation

The relativized inhabitation problem for $\mathsf{BCL}_k(\cap, \leq)$ is shown to be $(k+2)-$EXPTIME-complete by Düdder et al. [2012]. In [Düdder et al., 2012] an alternating Turing machine in Algorithm 3.1 is given deciding relativized inhabitation in $\mathsf{BCL}_0(\cap, \leq)$.

We use a shorthand notation for instruction sequences starting from existential states (CHOOSE ...) and instruction sequences starting from universal states (FORALL ...). A command CHOOSE $s \in \mathcal{S}$ branches from an existential state to successor states in which s gets assigned distinct elements of the set of successor states $\mathcal{S}$. A command FORALL$(l = 1 \dots m)$ seq_i branches from an universal state to m successor states from which each instruction sequence seq_i is executed.

Algorithm 3.1: ATM deciding inhabitation in $\mathsf{BCL}_0(\cap, \leq)$

1: *Input:* Γ, τ, k
2: *Output:* `INH1` accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$
3:
4: *loop:*
5: `CHOOSE` $(x : \sigma) \in \Gamma$;
6: $\sigma' := \bigcap\{S(\sigma) \mid S \text{ a substitution}\}$;
7: `CHOOSE` $m \in \{0, \dots \|\sigma'\|\}$;
8: `CHOOSE` $P \subseteq \mathbb{P}_m(\sigma')$;
9:
10: **if** $\bigcap_{\pi \in P} tgt_m(\pi) \leq \tau$ **then**
11: **if** $m = 0$ **then**
12: `ACCEPT`;
13: **else**
14: `FORALL` $(l = 1 \dots m)$ $\tau := \bigcap_{\pi \in P} arg_l(\pi)$;
15: `GOTO` *loop*;
16: **end if**
17: **else**
18: `FAIL`;
19: **end if**

Informally, the ATM proceeds as follows: τ holds the value of the current inhabitation goal. In lines 5–8, the machine non-deterministically guesses x, m, and P. Line 10 filters these guesses, checking whether the resulting target intersection is a subtype of the current goal type τ. If this is the case, the machine transitions to a universal state (line 14) from which every branch of

the computation spawns a new goal τ based on the corresponding argument types. Then, computation loops back to line 4. The computation stops when no further arguments have to be processed (line 12). The machine fails if no choices leading to acceptance are possible.

The machine can be shown to operate in alternating exponential space and with the results of Chandra et al. [1981] in $(k+2)$-EXPTIME [Düdder et al., 2012]. A short applicative example will show an inhabitation more detailed.

Example 3.3. *We will explain the algorithm by discussing an example in simple types without type variables. Assume a type environment*

$$\Gamma = \{x \ : \ \tau', y \ : \ \tau'', \ldots\}$$

and a relativized inhabitation question:

$$\Gamma \vdash ? \ : \ \tau$$

The algorithm in Figure 3.1 proceeds as follows

1. *Guess non-deterministically a combinator x out of Γ*

$$x \ : \ \tau_1 \to \ldots \to \tau_{n-1} \to \tau_n$$

2. *Guess non-deterministically, m, the number of arguments of x*

$$x \ : \ \underbrace{\tau_1 \to \ldots \to \tau_m}_{m \text{ arguments}} \to \underbrace{\tau_{m+1} \to \ldots \to \tau_n}_{target}$$

3. *Check if the target is a subtype of τ (else redo guesses)*

$$x \ : \ \tau_1 \to \ldots \to \tau_m \to \underbrace{\tau_{m+1} \to \ldots \to \tau_n}_{target \leq \tau ?}$$

4. *Check if* all *m arguments can be inhabited (else redo guesses)*

$$x \ : \ \underbrace{\tau_1 \to \ldots \to \tau_m}_{m \text{ arguments}} \to \tau_{m+1} \to \ldots \to \tau_n$$

Intersections types need a more sophisticated treatment.

- Line 10*: Targets of possible paths π where the intersection of all targets must be a subtype of τ.*

- Line 14*: Additionally, in order that τ is inhabited every argument of a possible path π must also be inhabited such that intersections of all corresponding arguments of possible paths π are inhabited as well.*

It can be seen that the algorithm tries to prove a target by proving all its arguments. In this case, such an algorithm is similar to logic programming with Horn-clauses.

A Horn-clause in reverse form:

$$q \longleftarrow p_1, \ldots, p_k.$$

with q as head of the Horn-clause and a tail $p_1, \ldots, p_k$. The selective linear definite (SLD) *resolution uses a similar proof strategy. In order to prove predicate q all predicates p_i with $i \in \{1, \ldots, k\}$ must be proven.*

In Prolog the previous Horn-clause can be written as:

$$q \text{ :- } p_1, \ldots, p_k.$$

In addition, a coding for a two-counter automaton in simple types is provided by Rehof [2013] and demonstrates the computational power even of simple types, because two-counter automatons are known to be Turin-complete.

3.6 Combinatory Logic Synthesis

Rehof [2013] proposed the *combinatory logic synthesis* (CLS) as a method that uses relativized type inhabitation in combinatory logic, for instance, in $\mathsf{BCL}_k(\cap, \leq)$ for composition synthesis. In this method, the type environment Γ can be understood as a *repository* of named components. A component x is specified via its intersection type σ and is included as typed combinator in the type environment or repository $(x : \sigma) \in \Gamma$. Intersection types can be used to build feature vectors by intersecting these features in order to refine specifications of components.

With a given repository Γ and a target specification τ, we construct an inhabitant e satisfying $\Gamma \vdash e : \tau$. A combinatory inhabitant e is a program composed by applicative combination of the components in the repository Γ. A type inhabitation algorithm could therefore be used to *compute* such programs e. It allows the automatic *synthesis* of the program e from a specific, given Γ that may be changed for different applications or scenarios. The framework of relativized inhabitation in combinatory logic might be the right logical model for this process, because the repository Γ may change. The idea

of combinatory logic synthesis is loosely connected to the concept of abstract logic programming languages presented by Miller et al. [1991] where a proof search defines the declarative semantic of a logic program.

3.7 Related Work on Synthesis

Since the pioneering work of Church [1957] on circuit synthesis inaugurated a class of deep lying problems in computer science (now commonly referred to as "Church's Problem"), automatic synthesis of software has been and remains an important active area of research. Some recent work on synthesis will be discussed.

In its most general form, the synthesis problem asks to automatically construct a program satisfying a given logical specification. The two major challenges in synthesis are

1. algorithmic complexity and
2. specification complexity.

Whereas synthesis has so far been employed in specialized fields of engineering (by way of example, circuit synthesis, controller synthesis, VLSI design), much potential still lies untapped in extending the scope of synthesis to larger classes of software-intensive applications. Currently, the first mentioned challenge (1) is being addressed by considering synthesis from a given library of components rather than synthesizing a program from scratch (for instance, current work by Lustig and Vardi [2009]). This line of work has, however, remained within the classical specification framework of temporal logic which is often difficult to use in practice for larger classes of software systems. Basin et al. [2004] survey different methods for synthesizing recursive programs in computational logic. Recently, Bodik and Jobstmann [2013] provide a concise introduction into algorithmic program synthesis and distinguish reactive synthesis as well as functional synthesis. Reactive synthesis focuses on synthesizing reactive systems by using temporal specifications, for example *linear time logic* (LTL). Whereas functional synthesis is characterized by candidate spaces. Candidate spaces are a set of programs that a synthesizer considers when deriving a program satisfying a given specification. Furthermore, such candidate spaces can discriminated into semantical (with axioms) or syntactical (with grammars) candidate spaces. Within the classification scheme provided in [Bodik and Jobstmann, 2013], combinatory logic synthesis can be classified as functional synthesis with semantic candidate spaces.

In their work, the research group of Rehof are developing new synthesis algorithms and specification techniques based on combinatory logic and

constructive type theory in [Rehof and Urzyczyn, 2011a], [Rehof and Urzyczyn, 2012], [Rehof, 2013], and [Düdder et al., 2012]. This allows specifications to be much easier to connect to components by exploiting that component interfaces are natively equipped with type structure and that type structure naturally allows scaling the level of abstraction in specifications. The basic approach is to specify components by means of taxonomic type structures in a semantic type system based on the formalism of intersection types, which is known to combine extreme simplicity with enormous semantic expressive power. While radically novel, this line of work has already demonstrated its viability in applications to robotics and the generation of GUI-applications, for example, presented in [Düdder et al., 2012].

Interestingly, the bounded problem for $k = 0$ is comparable to other synthesis frameworks, for instance, LTL- [Vardi, 2008] and propositional dynamic logic synthesis [Lange and Lutz, 2005, Tuominen, 1988], which are 2-EXPTIME-complete but appear to be algorithmically quite different. Our approach can be broadly compared in spirit (rather than in technology) to synthesis of loop free programs proposed by Gulwani et al. [2011] and proof-theoretic synthesis by Srivastava et al. [2010]. The combinatory approach is fundamentally different, at a technical level, from such approaches that are based on either temporal logic, automata theory, or traditional program logics. The specification language, the logic and the algorithmics of combinatory logic synthesis are distinct in being based on type theoretical and proof theoretical ideas for propositional logics of the Hilbert type. We expect that future work will raise many questions concerning engineering Hilbert-style propositional logics which is not as well understood as it should be. A general introduction to the combinatory standpoint can be found in [Rehof, 2013].

The CLS approach is related to adaptation synthesis via proof counting discussed by Haack et al. [2002], Wells and Yakobowski [2005], where semantic types are combined with proof search in a specialized proof system. In particular, we follow the approach of Haack et al. [2002], Wells and Yakobowski [2005] in using semantic specifications at the interface level. The idea of adaptation synthesis by Haack et al. [2002] is related to our notion of composition synthesis presented in Section 1.1 on page 3, however our logic is different, our design of semantic types with intersection types is novel, and the algorithmic methods are different. In [Haack et al., 2002] the specification language used is a typed predicate logic. Semantic intersection types can be compared to refinement types by Freeman and Pfenning [1991], but semantic types do not need to stand in a refinement relation to implementation types (as can be seen from our examples, this is important). Still, refinement types are a great source of inspiration for how semantic types can be used in specifications in many interesting situations.

Composition synthesis based on combinatory logic as presented by Hindley and Seldin [2008] with intersection types by Barendregt, Coppo, and Dezani-Ciancaglini [1983] was introduced and developed in the following line of work in [Rehof and Urzyczyn, 2011a], [Düdder et al., 2012], [Düdder et al., 2012], [Rehof, 2013], and [Düdder et al., 2013a]. The complexity theoretical and algorithmic foundations of the relativized inhabitation problem were laid down in [Rehof and Urzyczyn, 2011a, Düdder et al., 2012] (EXPTIME-completeness for the monomorphic restriction, $(k+2)$-EXPTIME-completeness for the k-bounded restrictions). The lower-bound techniques provide insights into the expressive power of inhabitation in the combinatory framework, including connections to tree automata in [Rehof and Urzyczyn, 2011a] and ATMs in [Düdder et al., 2012]. A type environment can be interpreted as logic program and relativized type inhabitation as execution of such a logic program. Grossof et al. [2003] present an approach for combining description logic programs that are related to description Horn logic.

A closely related work to component-based synthesis is presented by Steffen et al. [1997]. In that work linear process models, a sequence of components, are automatically synthesized from semantically extended temporal constraints in *semantic linear time logic* (SLTL), specifying a given set of components. This work also allows for semantic specification and uses types respectively subtyping to capture taxonomic hierarchies.

An orthogonal approach for synthesis of services is presented by Jannach and Leopold [2007]. In this work, an *artificial intelligence* (AI) planning algorithm, the *Stanford Research Institute Problem Solver* (STRIPS), is used to synthesize a pipeline of processors for transforming video data depending on display capabilities and users' preferences. Another approach is synthesis using constraint-solver as presented by Lamprecht et al. [2011]. The constraint solver searches for solutions in the solution space spanned by the constraint definitions.

Chapter 4

Optimization of CLS

This chapter discusses two orthogonal families of approaches for optimizing relativized inhabitation in $\mathsf{BCL}_k(\cap, \leq)$.

The first family of approaches addresses the optimization of the theoretical algorithm presented as ATMs in Chapter 3.3. These optimizations will lead to reformulations of the inhabitation algorithm including a combined heuristic optimization providing a significant speed-up. The combined heuristic uses sufficient restrictions on possible substitutions examined by the algorithm as well as a sufficient restriction on newly generated inhabitation requests. Employing the combined heuristic yields a speed-up that is verified by experiments later in this chapter.

The second family of approaches, considered in Section 4.2, address a possible implementation of the (distributed) relativized inhabitation algorithm. This family includes algorithmic optimization of a more general nature and are generally applicable to implementations of ATMs. With a distributed implementation in mind, a specialized shared data-structure, execution graph, for controlling and optimizing the computation of relativized inhabitation is presented. A group of optimizations on type term level simplifying types for efficient processing are discussed. Subsequently, computation of relativized inhabitation is optimized by various approaches manipulating the execution graph in order to reduce unnecessary computations. We will also apply results from type theory (for instance using Lemma 3.3.1 on page 42) to make these heuristical optimizations even more effective and efficient.

A distributed and concurrent algorithm unifying previous results of both families of optimization will be shown and discussed in Section 4.3.

4.1 Theoretical Algorithm

We begin with the optimizations of the theoretical algorithm by primarily exploiting type theoretic properties of relativized type inhabitation in $\mathsf{BCL}_k(\cap, \leq)$.

4.1.1 Restricting Intersections in Substitutions

One profound cause of the exponentially growing complexity of inhabitation in the logic $\mathsf{BCL}_k(\cap, \leq)$ (compared to the monomorphic restriction, $\mathsf{FCL}(\cap, \leq)$, presented in [Rehof and Urzyczyn, 2011a]) lies in the need to search for suitable instantiating substitutions S in rule (var). In [Düdder et al., 2012] it is shown that one needs only to consider rule (var) restricted to substitutions of the form $S : \mathit{Var}(\Gamma) \to \mathbb{T}_k(\mathit{At}_\omega(\Gamma, \tau))$, where $\mathit{At}_\omega(\Gamma, \tau)$ denotes the set of atoms occurring in Γ or τ, together with ω, and for $k \geq 0$, $\mathbb{T}_k$ denotes the set of all level-k types out of $\mathbb{T}$. This finitizes the inhabitation problem and immediately leads to decidability. Now, given a number $k \geq 0$, an environment Γ and a type τ, define for each x occurring in Γ the set of substitutions $\mathcal{S}_x^{(\Gamma,\tau,k)} = \mathit{Var}(\Gamma(x)) \to \mathbb{T}_k(\mathit{At}_\omega(\Gamma, \tau))$. This set, as well as the type size of substitutions, grows exponentially with k.

Let $\exp_k$ be the iterated exponential function, given by the following definition

$$\exp_0(n) = n$$
$$\exp_{k+1}(n) = 2^{\exp_k(n)}$$

The lemma below can be shown by an elementary counting argument.

Lemma 4.1.1 (Lemma 13 in [Düdder et al., 2012])**.** *For every $k \geq 0$, there is a polynomial $p(n)$ such that the number of level-k types over n atoms is at most $\exp_{k+1}(p(n))$, and the size of such types is at most $\exp_k(p(n))$. The number and size of* simple *level-k types (for a fixed k) is respectively bounded by a polynomial and a constant.*

The root of the $(k+2)$-Exptime-hardness result of Düdder et al. [2012] for inhabitation in $\mathsf{BCL}_k(\cap, \leq)$ is the fact that one cannot bypass, in the worst case, exploring such vast spaces of types and substitutions. However, in applications, as presented in [Düdder et al., 2012] and in [Düdder et al., 2012], it is to be expected that a complete, brute-force exploration of the sets $\mathcal{S}_x^{(\Gamma,\tau,k)}$ is unnecessary. This is the point of departure for our heuristic optimizations of the type theoretical algorithm of Düdder et al. [2012], which, for convenience, is stated in the following. It is a $(k+1)$-Expspace ATM, yielding a $(k+2)$-Exptime decision procedure.

Algorithm 4.1: ATM deciding inhabitation in $\mathsf{BCL}_k(\cap, \leq)$

Input: Γ, τ, k
Output: `INH1` accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$

loop:
`CHOOSE` $(x : \sigma) \in \Gamma$;
$\sigma' := \bigcap\{S(\sigma) \mid S \in \mathcal{S}_x^{(\Gamma,\tau,k)}\}$;
`CHOOSE` $n \in \{0, \ldots \|\sigma'\|\}$;
`CHOOSE` $P \subseteq \mathbb{P}_n(\sigma')$;

if $\bigcap_{\pi \in P} tgt_n(\pi) \leq \tau$ **then**
  if $n = 0$ **then**
    `ACCEPT`;
  else
    `FORALL` $(j = 1 \ldots n)$ $\tau := \bigcap_{\pi \in P} arg_j(\pi)$;
    `GOTO` *loop*;
  end if
else
  `FAIL`;
end if

The idea behind our first optimization is to show that for a type $\tau = \bigcap_{i \in I} \tau_i$ to satisfy $\tau \leq \sigma$, where $\sigma = \bigcap_{j \in J} \sigma_j$, the size of the index set I can be bounded by the size of the index set J of σ. One might at first conjecture that it always suffices to consider an index set I where $|I| \leq |J|$. This is not true as can easily be seen by considering $(a \to b) \cap (a \to c) \leq a \to (b \cap c)$, for example. In this example, only choosing both paths $a \to b$ and $a \to c$ altogether leads to inhabitation. But we show that the property holds for organized types.

Based on Lemma 3.2.2 on page 38 we characterize the subtypes of a path (generalizing Lemma 3 in [Düdder et al., 2012]):

Lemma 4.1.2. *Let $\tau = \bigcap_{i \in I} \tau_i$ where the τ_i are paths and let $\sigma = \beta_1 \to \ldots \to \beta_n \to p$ where $p \neq \omega$ is an atom.*

We have $\tau \leq \sigma$ if and only if there is an $i \in I$ with $\tau_i = \alpha_1 \to \ldots \to \alpha_n \to p$ and $\beta_j \leq \alpha_j$ for all $j \leq n$.

Proof. We write $\bigcap_{i \in I} \tau_i = \bigcap_{j \in J} a_j \cap \bigcap_{k \in K} \sigma_k \to \sigma'_k$ (in particular $I = J \cup K$).

$\Rightarrow$: We use induction over n.

$n = 0$: We have $\bigcap_{i \in I} \tau_i \leq p$ where p is a type constant. Lemma 3.2.2 on page 38 states that there must be a $j \in J$ with $a_j = p$.

$n \Rightarrow n+1$**:** Assume $\bigcap_{i\in I} \tau_i \leq \beta_1 \to \ldots \to \beta_{n+1} \to p$. Lemma 3.2.2 on page 38 further states that the set $H = \{k \in K | \beta_1 \leq \sigma_k\}$ is non-empty and $\bigcap_{h\in H} \sigma'_k \leq \beta_2 \to \ldots \to \beta_{n+1} \to p$. Note that each of the σ'_k is again a path. Therefore, we may apply the induction hypothesis to the last inequality and we see that there is some $h_0 \in H$ with $\sigma'_{h_0} = \alpha_2 \to \ldots \to \alpha_{n+1} \to p$ where $\beta_l \leq \alpha_l$ for all $2 \leq l \leq n+1$. Because $h_0 \in H$ we know that $\beta_1 \leq \sigma_{h_0}$. Setting $\alpha_1 = \sigma_{h_0}$, the type $\sigma_{h_0} \to \sigma'_{h_0} = \alpha_1 \to \ldots \to \alpha_{n+1} \to \tau$ has the desired properties.

$\Leftarrow$**:** For this direction we first show by induction over n that a type $\alpha_1 \to \ldots \to \alpha_n \to p$ with $\beta_j \leq \alpha_j$ for all $j \leq n$ is a subtype of $\beta_1 \to \ldots \to \beta_n \to p$.

$n = 0$**:** There is nothing to prove because both types are equal to p.

$n \Rightarrow n+1$**:** We want to show $\alpha_1 \to \ldots \to \alpha_{n+1} \to p \leq \beta_1 \to \ldots \to \beta_{n+1} \to p$. Because of Lemma 3.2.2 on page 38 this inequality holds if and only if $\beta_1 \leq \alpha_1$ and $\alpha_2 \to \ldots \to \alpha_{n+1} \to p \leq \beta_2 \to \ldots \to \beta_{n+1} \to p$. The first inequality holds by the assumption and the second one holds because of the induction hypothesis.

By assumption there is an $i \in I$ with $\tau_i = \alpha_1 \to \ldots \to \alpha_n \to p$ and $\beta_j \leq \alpha_j$ for all $j \leq n$. From the above we know $\tau_i \leq \beta_1 \to \ldots \to \beta_n \to p$ and therefore also $\bigcap_{i\in I} \tau_i \leq \beta_1 \to \ldots \to \beta_n \to p$.

□

For $\sigma = \bigcap_{j\in J} \sigma_j$ (not necessarily organized), it is easy to see that one has $\tau \leq \sigma$ iff for all j we have $\tau \leq \sigma_j$. Using this observation together with Lemma 4.1.2 we obtain:

Lemma 4.1.3. *Let $\tau = \bigcap_{i\in I} \tau_i$ and $\sigma = \bigcap_{j\in J} \sigma_j$ be organized types that are reduced with respect to ω.*

We have $\tau \leq \sigma$ iff there exists $I' \subseteq I$ with $|I'| \leq |J|$ and $\bigcap_{i\in I'} \tau_i \leq \sigma$.

Proof. The right-to-left implication is obvious.

Assume $\tau \leq \sigma$. This implies $\tau \leq \sigma_j$ for all $j \in J$. Fix $j \in J$. σ is organized and we write $\sigma_j = \beta^j_1 \to \ldots \to \beta^j_{n_j} \to p^j$. By the "if"-part of Lemma 4.1.2 on the preceding page there is an index $i_j \in I$ such that $\tau_{i_j} = \alpha^{i_j}_1 \to \ldots \to \alpha^{i_j}_{n_j} \to p^j$ with $\beta^j_k \leq \alpha^{i_j}_k$ for all $1 \leq k \leq n_j$. Set $I' := \{i \in I | \exists j \in J \text{ with } i = i_j\}$. Clearly $|I'| \leq |J|$ holds. For every $j \in J$ the "only if"-part of Lemma 4.1.2 on the previous page shows $\bigcap_{i\in I'} \tau_i \leq \sigma_j$ because τ_{i_j} satisfies the condition stated. This implies $\bigcap_{i\in I'} \tau_i \leq \sigma$. □

As shown in [Düdder et al., 2012], the key to an algorithm matching the lower bound for $\mathsf{BCL}_k(\cap,\leq)$ is a *path lemma* [Düdder et al., 2012, Lemma 11] which characterizes inhabitation by the existence of certain sets of paths in instances of types in Γ. The following lemma is a consequence of Lemma 4.1.3 on the facing page and Lemma 11 in [Düdder et al., 2012].

Lemma 4.1.4. *Let $\tau = \bigcap_{i\in I}\tau_i$ be organized and let $x:\sigma\in\Gamma$.*
The following are equivalent conditions:

1. $\Gamma\vdash_k x\ e_1\dots e_m : \tau$
2. *There exists a set P of paths in $\mathbb{P}_m(\bigcap\{\overline{S(\sigma)} \mid S\in\mathcal{S}_x^{(\Gamma,\tau,k)}\})$ such that*
   (a) $\bigcap_{\pi\in P} tgt_m(\pi) \leq \tau$;
   (b) $\Gamma\vdash_k e_i : \bigcap_{\pi\in P} arg_i(\pi)$, *for all* $i\leq m$.
3. *There exists a set $\mathcal{S}\subseteq\mathcal{S}_x^{(\Gamma,\tau,k)}$ of substitutions with $|\mathcal{S}|\leq|I|$ and a set $P'\subseteq\mathbb{P}_m(\bigcap_{S\in\mathcal{S}}\overline{S(\sigma)})$ of paths with $|P'|\leq|I|$ such that*
   (a) $\bigcap_{\pi\in P'} tgt_m(\pi) \leq \tau$;
   (b) $\Gamma\vdash_k e_i : \bigcap_{\pi\in P'} arg_i(\pi)$, *for all* $i\leq m$.

Proof. The implication 1. $\Rightarrow$ 2. follows from Lemma 10 in [Düdder et al., 2012].

We prove 2. $\Rightarrow$ 3 : Let P be as in the condition, i.e., $\bigcap_{\pi\in P} tgt_m(\pi) \leq \tau$. Lemma 4.1.3 on the preceding page states that there is $P'\subseteq P$ with $|P'|\leq|I|$ and $\bigcap_{\pi\in P'} tgt_m(\pi) \leq \tau$. For each $\pi\in P'$ there exists $S_\pi\in\mathcal{S}_x^{(\Gamma,\tau,k)}$ such that $\pi\in\mathbb{P}_m(\overline{S_\pi(\sigma)})$. Define $\mathcal{S}=\{S_\pi|\pi\in P'\}$. It is clear that $|\mathcal{S}|\leq|I|$ and that $P'\subseteq\mathbb{P}_m(\overline{\bigcap_{S\in\mathcal{S}}S(\sigma)})$. Thus, 3.(a) holds. Fix $i\leq m$. Because $P'\subseteq P$ we have $\bigcap_{\pi\in P} arg_i(\pi)\leq\bigcap_{\pi\in P'} arg_i(\pi)$. Since we have $\Gamma\vdash_k e_i : \bigcap_{\pi\in P} arg_i(\pi)$, rule $(\leq)$ yields $\Gamma\vdash_k e_i : \bigcap_{\pi\in P'} arg_i(\pi)$. Therefore, 3.(b) also holds.

The implication 3. $\Rightarrow$ 1. follows from a suitable application of the type rules. □

We immediately get the following corollary.

Corollary 4.1.5 (Path Lemma)**.** *Let $\tau=\bigcap_{i\in I}\tau_i$ be an organized type and let $(x:\sigma)\in\Gamma$.*
The following are equivalent conditions:

1. $\Gamma\vdash_k x\ e_1\dots e_m : \tau$
2. *There exists a set $\mathcal{S}\subseteq\mathcal{S}_x^{(\Gamma,\tau,k)}$ of substitutions with $|\mathcal{S}|\leq|I|$ and a set $P\subseteq\mathbb{P}_m(\overline{\bigcap_{S\in\mathcal{S}}S(\sigma)})$ of paths with $|P|\leq|I|$ such that*

(a) $\bigcap_{\pi\in P} tgt_m(\pi) \leq \tau$;

(b) $\Gamma \vdash_k e_j : \bigcap_{\pi\in P} arg_j(\pi)$, *for all* $j \leq m$.

Algorithm 4.2 below is a direct implementation of the path lemma (Corollary 4.1.5 on the previous page) and therefore decides inhabitation in $\mathsf{BCL}_k(\cap, \leq)$.

Algorithm 4.2: `INH1'`(Γ, τ, k)

1: *Input:* Γ, τ, k — *wlog:* All types in Γ and $\tau = \bigcap_{i\in I} \tau_i$ are organized
2: *Output:* `INH1'` accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$
3:
4: *loop:*
5: `CHOOSE` $(x : \sigma) \in \Gamma$;
6: `CHOOSE` $\mathcal{S} \subseteq \mathcal{S}_x^{(\Gamma,\tau,k)}$ with $|\mathcal{S}| \leq |I|$;
7: $\sigma' := \bigcap\{S(\sigma) | S \in \mathcal{S}\}$;
8: `CHOOSE` $n \in \{0, \ldots \|\sigma'\|\}$;
9: `CHOOSE` $P \subseteq \mathbb{P}_n(\overline{\sigma'})$ with $|P| \leq |I|$;
10:
11: **if** $\bigcap_{\pi\in P} tgt_n(\pi) \leq \tau$ **then**
12: **if** $n = 0$ **then**
13: `ACCEPT`;
14: **else**
15: `FORALL` $(j = 1 \ldots n)$ $\tau := \bigcap_{\pi\in P} \overline{arg_j(\pi)}$;
16: `GOTO` *loop*;
17: **end if**
18: **else**
19: `FAIL`;
20: **end if**

Algorithm 4.1 is identical to Algorithm 4.2 but for the fact that it ignores the restrictions $|\mathcal{S}| \leq |I|$ (line 6 in Algorithm 4.2) and $|P| \leq |I|$ (line 9). It can be seen, from purely combinatorial considerations, that the optimization resulting from taking these bounds into account can lead to arbitrarily large speed-ups (when $|I|$ is relatively small, as can be expected in practice).

4.1.2 Intersection Type Matching

Considering the ATM in Figure 3.1 on page 44, the complexity arises from the intersection built in line 2 taking up exponential space. The $(k+2)-$EXPTIME-hardness result of Düdder et al. [2012] shows that this complexity cannot be

avoided in the worst case, but from a pragmatic point of view the ATM is *very* suboptimal. In line 2 *every* possible instantiation of σ is combined into one large intersection. From this intersection the paths relevant for inhabiting τ are singled out in line 4. If it is possible to filter out substitutions beforehand that do not contribute to inhabitation of τ the size of the intersection constructed in line 2 can be reduced. This can lead to drastic speedups.

A subtyping relation between a type resulting from instantiations of σ and τ has to be checked (see for line 5 in Figure 3.1 on page 44). Thus, an important opportunity for speedups by filtering out nonessential substitutions lies in solving the *intersection type matching problem* (cMATCH) [Düdder et al., 2013a], i.e., the problem whether, given σ' and τ', there exists a substitution S with $S(\sigma') \leq \tau'$. This problem is NP-complete [Düdder et al., 2013a]. We present a variant of the ATM above, which uses a decision-procedure for intersection type matching, $\texttt{Match}(\sigma' \leq \tau')$, to exclude unnecessary substitutions. We further use intersection type matching to exclude substitutions downstream which cannot lead to successful inhabitation of τ, *even though* the test in line 5 succeeds. This may be the case because new inhabitation questions are created which cannot be solved. The matching algorithm is shown in Algorithm 4.3 on the next page. This outlined algorithm $\texttt{Match}(C)$ does not handle covariant type constructors. Because the covariant type constructors are not distributive, $\texttt{Match}(C)$ can be extended to process these type constructors by including further case distinctions for recursively solving constraints occuring in arguments of a covariant type constructor.

Definition 4.1.1. ***Matchable***
Type σ is matchable with type τ if there exists a substitution S such that $S(\sigma) \leq \tau$ holds.

Extending this type matching algorithm by operating on $\leq_C$ might change the problem's complexity. This is not the case and is shown in the following corollary.

Corollary 4.1.6. *Deciding* cMATCH *and* CMATCH[1] *with* $\leq_C$ *is* NP*-complete.*

Proof. This follows directly from the proof of lemma 4 in [Düdder et al., 2013a] and the Lemma 3.2.1 on page 37 because arguments of covariant type constructors are bundle sets of constraints without introducing novel type constructors. □

Using Corollary 4.1.6, we can exchange CMATCH's subtyping relation without effecting a change in the complexity of algorithm $\texttt{Match}(C)$.

[1] In the sense of the definition given in [Düdder et al., 2013a].

Algorithm 4.3: `Match`(C) from Düdder et al. [2013a]

Input: $C = \{\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n\}$ such that for all i at most one of σ_i and τ_i contains variables. Furthermore, all types have to be organized.
Output: **true** if C can be matched otherwise **false**

while $\exists$ nonbasic constraint in C **do**
  choose a nonbasic constraint $c = (\tau \leq \sigma) \in C$
  reduce τ and σ with respect to ω
  switch
    case: c does not contain any variables
      if $\tau \leq \sigma$ **then**
        $C := C \backslash \{c\}$
      else
        return false
      end if
    case: $\sigma = \bigcap_{i \in I} \sigma_i$
      $C := C \backslash \{c\} \cup \{\tau \leq \sigma_i | i \in I\}$

    write $\tau = \bigcap_{i \in I} \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i$, $\sigma = \sigma_1 \to \ldots \to \sigma_m \to a$
    write $I_1 = \{i \in I | m_i < m\}$, $I_2 = \{i \in I | m_i = m\}$,
      $I_3 = \{i \in I | m_i > m\}$
    case: σ contains variables, τ does not contain any variables
      if $a \in \mathbb{V}$ **then**
        CHOICE:
          1.) $C := C \backslash \{c\} \cup \{\omega \leq a\}$
          2.) choose $\emptyset \neq I' \subseteq I_2 \cup I_3$
            $C := C \backslash \{c\} \cup \{\overline{\sigma_j} \leq \overline{\tau_{i,j}} | i \in I', 1 \leq j \leq m\} \cup$
            $\{\bigcap_{i \in I'} \tau_{i,m+1} \to \ldots \to \tau_{i,m_i} \to p_i \leq a\}$
      else
        choose $i_0 \in I_2$
        $C := C \backslash \{c\} \cup \{\overline{\sigma_j} \leq \overline{\tau_{i_0,j}} | 1 \leq j \leq m\} \cup \{p_{i_0} \leq a\}$
      endif
    case: τ contains variables, σ does not contain any variables
      choose $i_0 \in I_1 \cup I_2$
      $C := C \backslash \{c\} \cup \{\overline{\sigma_j} \leq \overline{\tau_{i_0,j}} | 1 \leq j \leq m_{i_0}\} \cup$
        $\{p_{i_0} \leq \sigma_{m_{i_0}+1} \to \ldots \to \sigma_m \to a\}$
  end switch
end while
if C is consistent **then**
  return true
else
  return false
end if

This algorithm is discussed in detail in [Düdder et al., 2013a]. Line 21 contains a non-deterministic choice.

4.1.3 Matching Optimization

Algorithm 4.4 below checks for every target of every component σ_j of σ and τ_i of the inhabitation goal τ whether the constraint consisting of the corresponding target and τ_i is matchable. The τ_i can be inhabited by different components of σ. The number n of arguments, however, has to be the same for all i. This condition leads to the construction of N in line 10. Note that we may need to compute $tgt_n(\sigma_j)$ in line 8 where n is larger than $m_j := \|\sigma_j\|$. Any such call to Algorithm 4.3 on the preceding page in this line is assumed to return false if $tgt_{m_j}(\sigma_j)$ is a type constant. If $tgt_{m_j}(\sigma_j) = \alpha$, then we check matchability of $\alpha \leq \tau_i$. The following lemma shows that it suffices to only consider n with $n \leq \|\sigma\| + k$ in lines 7 and 10.

Lemma 4.1.7. *Let σ be an organized type and let S be a level-k substitution. We have $\|S(\sigma)\| \leq \|\sigma\| + k$.*

Proof. Let $\sigma_1 \to \ldots \to \sigma_m \to \alpha$ be a longest path in σ whose target is a variable. Note that $m \leq \|\sigma\|$. The worst case is that $S(\alpha)$ is a path of length k. In this case we have $\|S(\sigma_1) \to \ldots \to S(\sigma_m) \to S(\alpha)\| = m + k$. There are two cases: If $S(\sigma_1) \to \ldots \to S(\sigma_m) \to S(\alpha)$ is a longest path in $S(\sigma)$, then we have $\|S(\sigma)\| = m + k \leq \|\sigma\| + k$. Otherwise, a longest path in $S(\sigma)$ must have been created by instantiating a longest path in σ, and such a path must be longer than m (and it does not have a variable in the target!). We conclude $\|S(\sigma)\| = \|\sigma\| \leq \|\sigma\| + k$. □

We need an adaptation of Lemma 4.1.5 on page 55 to prove the correctness of Algorithm 4.4 on the next page.

Lemma 4.1.8. *Let τ be a path and let $x : \sigma \in \Gamma$ where $\sigma = \bigcap_{j \in J} \sigma_j$ is organized.*

The following are equivalent conditions:

1. $\Gamma \vdash_k x\ e_1 \ldots e_m : \tau$
2. *There exists $S \in \mathcal{S}_x^{(\Gamma,\tau,k)}$ and a path $\pi \in \mathbb{P}_m(\overline{S_\tau(\sigma)})$ such that*
   (a) $tgt_m(\pi) \leq \tau$;
   (b) $\Gamma \vdash_k e_l : arg_l(\pi)$, *for all* $l \leq m$.
3. *There exists $j \in J$, $S \in \mathcal{S}_x^{(\Gamma,\tau,k)}$, and $\pi \in \mathbb{P}_m(\overline{S(\sigma_j)})$ such that*

Algorithm 4.4: $\texttt{INH2}(\Gamma, \tau, k)$

Input: Γ, τ, k — *wlog:* All types in Γ and $\tau = \bigcap_{i \in I} \tau_i$ are organized
Output: `INH2` accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$

loop:
`CHOOSE` $(x : \sigma) \in \Gamma$;
write $\sigma = \bigcap_{j \in J} \sigma_j$
for all $i \in I, j \in J, n \leq \|\sigma\| + k$ **do**
  $\texttt{candidates}(i, j, n) := \texttt{Match}(tgt_n(\sigma_j) \leq \tau_i)$
end for
$N := \{n \leq \|\sigma\| + k \mid \forall_{i \in I} \exists_{j \in J} : \texttt{candidates}(i, j, n) = \textbf{true}\}$
`CHOOSE` $n \in N$;
for all $i \in I$ **do**
  `CHOOSE` $j_i \in J$ with $\texttt{candidates}(i, j_i, n) = \textbf{true}$
  `CHOOSE` $S_i \in \mathcal{S}_x^{(\Gamma, \tau_i, k)}$
  `CHOOSE` $\pi_i \in \mathbb{P}_n(\overline{S_i(\sigma_{j_i})})$
end for

if $\forall_{i \in I} : tgt_n(\pi_i) \leq \tau_i$ **then**
  if $n = 0$ **then**
    `ACCEPT`;
  else
    `FORALL` $(l = 1 \ldots n)$ $\tau := \bigcap_{i \in I} \overline{arg_l(\pi_i)}$;
    `GOTO` *loop*;
  end if
else
  `FAIL`;
end if

(a) $tgt_m(\pi) \leq \tau$*;*

(b) $\Gamma \vdash_k e_l : arg_l(\pi)$*, for all* $l \leq m$*.*

Proof. The implication 1. ⇒ 2. follows from Corollary 4.1.5 on page 55.

We want to prove 2. ⇒ 3 : Denote by S' and π' the substitution and path in condition 2. Because $\mathbb{P}_m(\overline{S'(\sigma)}) = \mathbb{P}_m(\bigcap_{j \in J} \overline{S'(\sigma_j)})$ it is clear that there is an index j' such that π' occurs in $\overline{S'(\sigma_{j'})}$. Choosing $j = j'$, $S = S'$, and $\pi = \pi'$, the conditions clearly hold.

The implication 3. ⇒ 1. follows from a suitable application of the type rules. □

We get the following corollary:

Corollary 4.1.9. *Let $\tau = \bigcap_{i\in I} \tau_i$ be organized and let $(x : \sigma) \in \Gamma$ where $\sigma = \bigcap_{j\in J} \sigma_j$ is also organized.*
The following are equivalent conditions:

1. $\Gamma \vdash_k x\ e_1 \dots e_m : \tau$
2. *For all $i \in I$ there exist $j_i \in J$, $S_i \in \mathcal{S}_x^{(\Gamma,\tau_i,k)}$, and $\pi_i \in \mathbb{P}_m(\overline{S_i(\sigma_{j_i})})$ with*
   (a) $tgt_m(\pi_i) \leq \tau_i$;
   (b) $\Gamma \vdash_k e_l : \bigcap_{i\in I} arg_l(\pi_i)$, *for all $l \leq m$.*

Proof. It is clear that $\Gamma \vdash_k x\ e_1 \dots e_m : \tau$ is equivalent to $\Gamma \vdash_k x\ e_1 \dots e_m : \tau_i$ for all $i \in I$. An application of the equivalence of conditions *1.* and *3.* of Lemma 4.1.8 on page 59 shows that this is equivalent to the following condition: For all $i \in I$ there exist $j_i \in J$, $S_i \in \mathcal{S}_x^{(\Gamma,\tau_i,k)}$, and $\pi_i \in \mathbb{P}_m(\overline{S_i(\sigma_{j_i})})$ such that

1. $tgt_m(\pi_i) \leq \tau_i$;
2. $\Gamma \vdash_k e_l : arg_l(\pi_i)$, for all $l \leq m$.

With these choices this condition is equivalent to condition *2.* of the corollary. □

Algorithm 4.4 is a direct realization of condition *2.* of the corollary above. This proves the correctness of the algorithm. Again, a combinatorial consideration shows that this optimization can lead to very large speed-ups since it prevents the consideration of useless substitutions.

Based on a decision procedure for intersection type matching in [Düdder et al., 2013a], we discuss an optimization of the ATM in Figure 3.1 on page 44.

4.1.4 Lookahead Optimization

Using matching, we formulate a necessary condition for newly created inhabitation questions to be solvable: for such a question to be solvable, at least each $\bigcap_{i\in I} \overline{arg_l(\pi_i)}$ in line 22 in Algorithm 4.4 on the preceding page must be inhabited. The following example motivates this intuition.

Example 4.1. *Assume a type environment:*

$$\begin{aligned}\Gamma = \{ & \\ & A : \sigma_1 \to \sigma_2 \to \sigma_3 \to \tau, \\ & B : \sigma_4 \to \sigma_5 \to \sigma_6 \to \tau, \\ & C : \sigma_1, \\ & D : \sigma_4, \\ & E : \sigma_5, \\ & F : \sigma_6 \\ \} & \end{aligned}$$

In Figure 4.1 on the facing page a corresponding proof-graph for $\Gamma \vdash ? : \tau$ *is shown. We use circles to represent inhabitation questions and rectangles to represent choices of the relativized inhabitation algorithm that are related to an inhabitation question.*

To solve this inhabitation question, the algorithm chooses either $A : \sigma_1 \to \sigma_2 \to \sigma_3 \to \tau$ *with* $n = 3$ *or* $B : \sigma_4 \to \sigma_5 \to \sigma_6 \to \tau$ *also with* $n = 3$*, generating sub inhabitation questions for the arguments* σ_1*,* σ_2*, and* σ_3 *or respectively* σ_4*,* σ_5*, and* σ_6*.*

Each of these arguments must be inhabited with a suitable combinator that has a suitable σ_i *with* $i \in \{1, \ldots, 6\}$ *as a target. These combinators exist for* σ_1*,* σ_4*,* σ_5*, and* σ_6 *namely* C*,* D*,* E*, and* F*. As a consequence, the inhabitation request is only successful for the choice* $B : \sigma_4 \to \sigma_5 \to \sigma_6 \to \tau$ *with* $n = 3$ *whereas a choice of* $A : \sigma_1 \to \sigma_2 \to \sigma_3 \to \tau$ *with* $n = 3$ *fails.*

The initial inhabitation request $\Gamma \vdash ? : \tau$ *could have prevented three unnecessary inhabitation requests* $\Gamma \vdash ? : \sigma_1$*,* $\Gamma \vdash ? : \sigma_2$*, and* $\Gamma \vdash ? : \sigma_3$ *if it would have used a lookahead for inhabited arguments. The strategy of the lookahead optimization that the target type of an combinator as well as its arguments are checked for (possibly) being inhabited.*

We use the matching-algorithm to check a necessary condition for this to be possible. We refer to this optimization as the *lookahead* test.

Even though this necessary condition may seem rather coarse it turned out to have significant impact on the runtime. This impact will be discussed in Section 4.1.5 on page 66. We obtain the ATM presented in Figure 4.5 on page 64.

The FOR EACH ... DO-constructs in lines 3 and 7 describe FOR-loops and not universal states of the ATM. We point out a few details regarding the implementation of the matching-based optimizations. Matching is included in line 4 to check which type component σ_j of σ has a target with m arguments

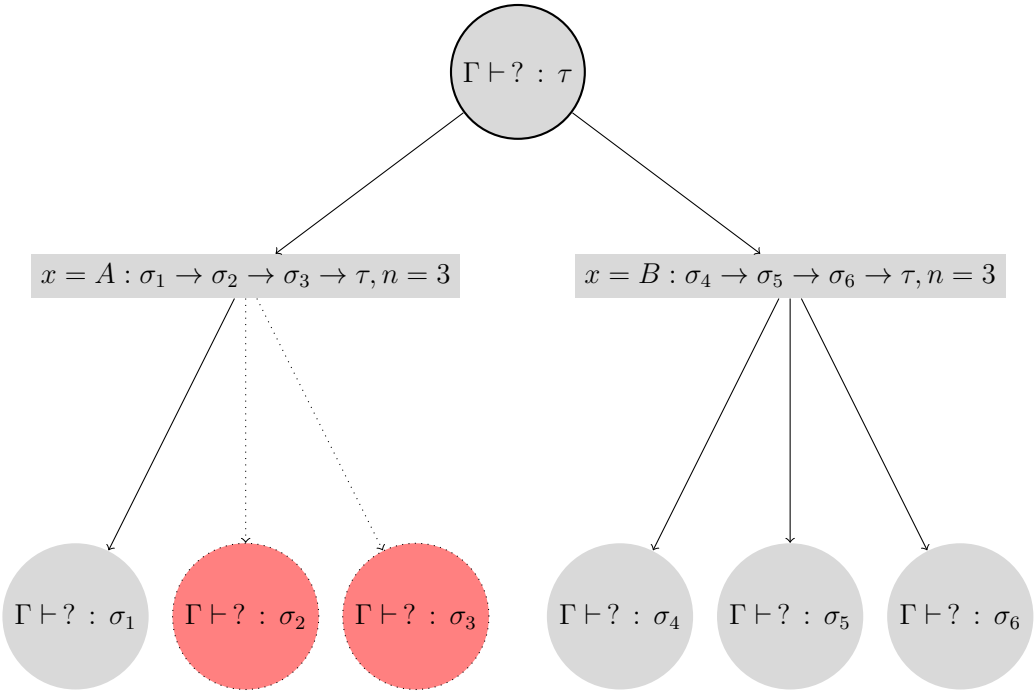


Figure 4.1: Example proof graph with lookahead

that can be instantiated such that it is a subtype of a fixed type component τ_i of τ. Executing these matching-checks, we store the results in the matrix `candidates`, i.e., `candidates`$(i, j, m) =$ **true** if and only if the constraint $tgt_m(\sigma_j) \leq \tau_i$ is matchable. Line 5 uses `candidates` to filter possible path lengths m: Fixing m, then m is a possible number of arguments, only if there exists j with `candidates`$(i, j, m) =$ **true** for all i. This is because every τ_i may be inhabited by a *different* type component σ_j of σ, if it is done with the *same* number of arguments. Having chosen a possible number of arguments in line 6, the FOR EACH ... DO-block from line 7 to 12 first chooses for every τ_i a type component σ_{j_i}, a substitution S_i, and a path π_i of length at least m in $S_i(\sigma_{j_i})$. Then, it first checks whether the choices are suitable to inhabit τ_i (i.e., whether $tgt_m(\pi_i) \leq \tau_i$). The lookahead-test is performed in lines 11 and 12. The idea is that a path π_i is only meaningful if *for all* $1 \leq l \leq m$ and *all* type components π' of $arg_l(\pi)$ there exists $(y : \rho) \in \Gamma$ such that ρ has a target $tgt_k(\rho)$ for some k for which `Match`$(tgt_k(\rho) \leq \pi')$ returns **true**. If there is no such $(y : \rho)$, then $arg_l(\pi)$ and thus $\bigcap_{\pi \in P} arg_l(\pi)$ cannot be inhabited.

The correctness of the optimized algorithm is based on the Corollary 4.1.9 on page 61 and lemma whose proofs can be found in [Düdder et al., 2012, Corollary 34 respectively Lemma 36]. The proposition characterizes the

Algorithm 4.5: ATM with lookahead-test

Input : Γ, τ— all types in Γ and $\tau = \bigcap_{i \in I} \tau_i$ organized
loop :
1 CHOOSE $(x : \sigma) \in \Gamma$;
2 write $\sigma \equiv \bigcap_{j \in J} \sigma_j$
3 FOR EACH $i \in I, j \in J, m \leq \|\sigma\|$ DO
4 $\quad$ $\texttt{candidates}(i, j, m) := \texttt{Match}(tgt_m(\sigma_j) \leq \tau_i)$
5 $M := \{m \leq \|\sigma\| \mid \forall_{i \in I} \exists_{j \in J} : \texttt{candidates}(i, j, m) = \textbf{true}\}$
6 CHOOSE $m \in M$;
7 FOR EACH $i \in I$ DO
8 $\quad$ CHOOSE $j_i \in J$ with $\texttt{candidates}(i, j_i, m) = \textbf{true}$
9 $\quad$ CHOOSE S_i a substitution
10 $\quad$ CHOOSE $\pi_i \in \mathbb{P}_m(\overline{S_i(\sigma_{j_i})})$ with $tgt_m(\pi_i) \leq \tau_i$ and
11 $\quad$ $\forall 1 \leq l \leq m \; \forall \pi' \in \overline{arg_l(\pi_i)} \; \exists (y : \rho) \in \Gamma \; \exists$ a path ρ'
12 $\quad$ in $\rho \; \exists k \; : \; \texttt{Match}(tgt_k(\rho') \leq \pi') = \textbf{true}$
13 IF $(m = 0)$ THEN ACCEPT;
14 ELSE FORALL$(l = 1 \ldots m)$
15 $\quad\quad$ $\tau := \bigcap_{i \in I} arg_l(\pi_i)$;
16 $\quad\quad$ GOTO *loop*;

condition under which an organized type τ is inhabited.

We need an auxiliary lemma for the desired lemma:

Lemma 4.1.10. *Let τ be a type and let $\rho = \rho_1 \to \ldots \to \rho_r \to a$ be a path. Let S be a substitution. Let $\pi \in \mathbb{P}_m(\overline{S(\rho)})$ such that $tgt_m(\pi) \leq \tau$.*

There exist $h \leq r$ and a substitution S' such that $S'(tgt_h(\rho)) \leq \tau$. Furthermore, $l(S') \leq l(S)$ and any constants occurring in the image of S' also occur in the image of S.

Proof. We distinguish two cases:

$a \in \mathbb{V}$**:** Write $a = \alpha$, i.e., $\rho_1 \to \ldots \to \rho_r \to \alpha$. Again we distinguish two cases:

$m \leq r$**:** Then we can write $tgt_m(\pi) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to \pi_{r+1} \to \ldots \to \pi_s \to b \leq \tau$, where $\pi_{r+1} \to \ldots \to \pi_s \to b$ is a component in $\overline{S(\alpha)}$. Choosing $h = m$ and $S' = S$, we get $S'(tgt_h(\rho)) = S(tgt_m(\rho)) = S(tgt_m(\rho)) = S(\rho_{m+1} \to \ldots \to \rho_r \to \alpha) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to S(\alpha) \leq S(\rho_{m+1}) \to \ldots \to$

$S(\rho_r) \to \pi_{r+1} \to \ldots \to \pi_s \to b \leq \tau$ where the first inequality follows because $\pi_{r+1} \to \ldots \to \pi_s \to b$ is a component in $\overline{S(\alpha)}$.

$m > r$: Then, we can write $\pi = S(\rho_1) \to \ldots \to S(\rho_r) \to \pi_{r+1} \to \ldots \to \pi_s \to b$, where $\pi_{r+1} \to \ldots \to \pi_s \to b$ is a component in $\overline{S(\alpha)}$. We get $tgt_m(\pi) = \pi_{m+1} \to \ldots \to \pi_s \to b \leq \tau$. We choose $h = r$ and we define $S' = \{\alpha \mapsto \pi_{m+1} \to \ldots \to \pi_s \to b\}$. Then we have $S'(tgt_h(\rho)) = S'(\alpha) = \pi_{m+1} \to \ldots \to \pi_s \to b \leq \tau$.

$a \notin \mathbb{V}$: We have $\overline{S(\rho)} = S(\rho_1) \to \ldots \to S(\rho_r) \to a$. Thus, $m \leq r$ and $tgt_m(\pi) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to a \leq \tau$. Choosing $h = m$ and $S' = S$, we get $S'(tgt_h(\rho)) = S(tgt_m(\rho)) = S(\rho_{m+1} \to \ldots \to \rho_r \to a) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to a \leq \tau$.

It is clear that $l(S') \leq l(S)$ and that the statement about the constants in the image of S' holds. □

The lemma formulates the necessary condition whose violation is checked in lines 11 and 12 of the ATM. Using the fact that any term e can be written as $e = x\ e_1 \ldots e_m$, the lemma follows from the only if-direction of the proposition.

Lemma 4.1.11. *Assume $\Gamma \vdash e : \tau$, where τ is a path.*

There exist $(y : \rho) \in \Gamma$, a path ρ' in ρ, $k \leq \|\rho'\|$ and a substitution S such that $S(tgt_k(\rho')) \leq \tau$.

Proof. We write $e = x\ e_1 \ldots e_m$ and we use the only-if direction of Corollary 4.1.9 on page 61 to conclude that there exists an $(x : \sigma) \in \Gamma$, $j \in J$, $S \in \mathcal{S}_x^{(\Gamma,\tau,k)}$, and $\pi \in \mathbb{P}_m(\overline{S(\sigma_j)})$ with $tgt_m(\pi) \leq \tau$. Thus, setting $(y : \rho) = (x : \sigma)$, we know that there exist $(y : \rho) \in \Gamma$, a path ρ' in ρ (note that $\rho' = \sigma_j$), $S \in \mathcal{S}_x^{(\Gamma,\tau,k)}$, and $\pi \in \mathbb{P}_m(\overline{S(\rho')})$ with $tgt_m(\pi) \leq \tau$. Applying Lemma 4.1.10 on the preceding page we get the statement of the lemma. □

Correctness follows because the optimized ATM directly implements the condition of Proposition 4.1.9 on page 61 and combines it for every path τ_i in τ with the necessary condition of Lemma 4.1.11. Line 10 of the ATM incorporates the check of $tgt_m(\pi_i) \leq \tau_i$ (condition 2.(a) of Proposition 4.1.9 on page 61) into the choice of π_i. This makes the **if**-block in line 5 of the ATM in Figure 3.1 on page 44 obsolete.

4.1.5 Experimental Evaluation

In order to evaluate the impact of the heuristic and optimization by the previous finding, we can define the following question:

Can the improvement of the optimized algorithm compared to the original algorithm be quantified as a function of the arguments of an inhabitant and the type constants that are present in the inhabitation problem respectively the type environment by a parameterized problem?

In this experiment, n and m will determine the number of arguments respectively the number of type constants that are present in a type environment.

Setup

We implemented both ATMs in Figures 3.1 on page 44 and 4.5 on page 64 in our tool called (CL)S that will be discussed later in Section 5 on page 105. We start by describing the experimental setup: We used F# and C# (.NET Framework 4.5) to implement the ATMs. The core algorithms realizing the respective ATMs were implemented in F# whereas the control-algorithm was implemented in C#. The complete setup configuration is included in Appendix B on page 207 in Subsection B.3.1.

The experimental example does arithmetic in $\mathbb{Z}_n$. This example is qualified for answering the research question, because every combinator occurring in the type environment has exactly m arguments and the number of type constants is n. This allows a fine grained control of the occurring inhabitation questions and the generated substitutions. All inhabitation questions are uniform in the number of arguments and the substitutions share the same range (over the same type constants).

We exploit the fact that finite function tables can be coded by means of intersection types by Barendregt et al. [1983]. The successor function in $\mathbb{Z}_3$, for example, can be coded by the combinator $s_3 : (0 \to 1) \cap (1 \to 2) \cap (2 \to 0)$. Similarly, we may define combinators i_3 and p_3 with corresponding types representing the identity respectively the predecessor function in $\mathbb{Z}_3$. We consider i_3, s_3, and p_3 as our basic building blocks from which we want to synthesize other functions in $\mathbb{Z}_3$ by function composition. For this purpose we introduce a composition combinator $c : (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$ into the repository. It is a combinator that takes two functions as arguments and returns a function representing their composition. Note that the combinators i_3, s_3, and p_3 have monomorphic types (0, 1, and 2 are type constants), whereas c is polymorphically typed (type variables may be instantiated under substitutions). Considering a repository Γ containing i_3, s_3, p_3, and c as

above, we ask the inhabitation question $\Gamma \vdash ? : (0 \to 2) \cap (1 \to 0) \cap (2 \to 1)$, i.e., we want to synthesize addition of 2 in $\mathbb{Z}_3$. Using the rules in Figure 3.1 on page 41, one obtains the three inhabitants $c(s_3, s_3)$, $c(i_3, p_3)$, and $c(p_3, i_3)$. This is not surprising, since addition of 2 can be realized in $\mathbb{Z}_3$ by adding 1 twice *or* by subtracting 1.

We chose this example for our experiments since it scales along two parameters. We may increase n or the arity m of c as a function taking other functions as an argument. The combinator $c_m : (\alpha_0 \to \alpha_1) \to \ldots \to (\alpha_{m-1} \to \alpha_m) \to (\alpha_0 \to \alpha_m)$ takes m functions as an argument and returns their sequential composition. We obtain the following general form of Γ_n^m:

$$\begin{aligned}
&\{i_n : f \cap (0 \to 0) \cap \ldots \cap \big((n-1) \to (n-1)\big),\\
&\ s_n : f \cap (0 \to 1) \cap \ldots \cap \big((n-1) \to 0\big),\\
&\ p_n : f \cap \big(0 \to (n-1)\big) \cap \ldots \cap \big((n-1) \to (n-2)\big),\\
&\ c_m : \big(f \cap (\alpha_0 \to \alpha_1)\big) \to \ldots \to \big(f \cap (\alpha_{m-1} \to \alpha_m)\big) \to\\
&\qquad (\alpha_0 \to \alpha_m)\}
\end{aligned}$$

We enriched the types of the basic combinators by adding a type constant f, which indicates that the combinator is a function. The arguments of c_m are also specified to be functions. This is necessary to prohibit self-application of c_m which would lead to cyclic solutions.[2]

Execution

Γ_n^m is very well suited for an experimental evaluation since we may trace the effects that result from varying the parameters. We use it to compare our implementations of both ATMs. In the experiments we asked the inhabitation question $\Gamma_n^m \vdash ? : (0 \to 2) \cap (1 \to 3) \cap \ldots \cap \big((n-1) \to 1\big)$, i.e., we synthesized addition of 2 in $\mathbb{Z}_n$. Our main interest was in comparing the speedups achieved by the lookahead-test compared to the unoptimized ATM. We use the number of inhabitation questions created as measure for the runtime because this number determines the execution time of the algorithm linearly since the execution time per inhabitation question is more or less constant for fixed n and m. We had to estimate this number[3] for some values of n and m for the unoptimized implementation because computation exceeded resources. We do not measure memory consumption per inhabitation question because it is also more or less constant for fixed n and m. This is because substitutions are not instantiated up-front but as needed.

[2]This technique corresponds to the semantic specification of software components in order to control synthesis.

[3]Estimates are indicated by an asterisk, *, in Table 4.1 on page 69.

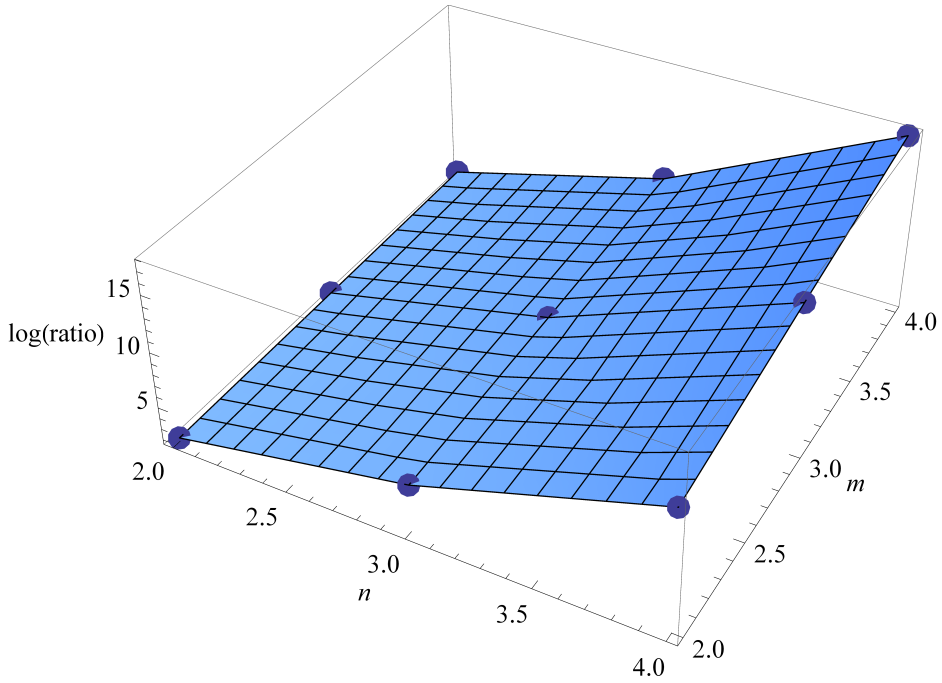


Figure 4.2: Logarithmic Plot of Ratio of Created Inhabitation Questions

We analyze the speedup by benchmarking the performance of the optimized algorithm with respect to the unoptimized algorithm. We graphically depict our experimental results by a 3D-plot depicted in Figure 4.2. We plotted the logarithm of the ratio of the number of created inhabitation questions for the unoptimized implementation divided by the corresponding number of the lookahead-implementation against n and m. Thus, we normalized the numbers of created inhabitation questions to the unoptimized algorithm. Hence, a larger ratio in these charts corresponds to a greater speedup achieved by the lookahead. In Figure 4.2 the discrete values are indicated by the bullets. We interpolated these values in order to illustrate the increase in the speedup.

Results

Table 4.1 on the next page presents some absolute figures. For both implementations and the respective values of n and m it lists the number of created inhabitation questions (respectively the estimates for the unoptimized ATM), #I, and the mean runtime of the algorithms in milliseconds, $\overline{\text{RT}}$/ms, over runs of five experiments. For $n = m = 2$ the unoptimized implementation is almost as fast as the lookahead-ATM even though it has to create 8 times as

		Unoptimized ATM		Lookahead-ATM	
n	m	#I	$\overline{\text{RT}}$/ms	#I	$\overline{\text{RT}}$/ms
2	2	73	112.2	9	109.8
2	3	973	209.6	49	111.4
2	4	11 665	2245.2	257	124.4
3	2	43 905	12 504	55	124
3	3	$1.4 \times 10^{10*}$	–	2188	354
3	4	$4.8 \times 10^{12*}$	–	78 733	50 959.6
4	2	$1.3 \times 10^{14*}$	–	33	142.4
4	3	$6.6 \times 10^{18*}$	–	3889	1314.6
4	4	$3.3 \times 10^{23*}$	–	640 001	7.5×10^6

Table 4.1: Experimental Data for Γ_n^m

many inhabitation questions. These results from the fact that the lookahead-optimization requires some computational effort which for small values of n and m may be significant. However, for larger n and m the improvement is obvious. The estimated numbers of inhabitation goals for the unoptimized ATM are only *very* rough lower bounds. An estimate for this number was not feasible directly since the number of substitutions got so large that it was impossible to estimate the size of the organization of the intersection created in line 2 of the unoptimized ATM out of which subsets of paths have to be chosen (line 4). This size, however, determines the number of new inhabitation questions that have to be checked in line 9. Instead, we only estimated the inhabitation questions created by an ATM in Algorithm 4.4 on page 60 that only implements the condition of Proposition 4.1.9 on page 61 (without the lookahead check). Such an ATM would only choose *one* substitution per type component of the inhabitation goal which is already an enormous improvement over the unoptimized ATM, and therefore such an estimate may serve as a lower bound for the number of inhabitation questions created by the unoptimized ATM. Nonetheless, the estimates show that the implementation of the unoptimized ATM is completely infeasible even for small values of n and m.

Example 4.2. *We consider the case $n = 4$ and $m = 3$ in more detail by estimating the possible number of substitutions. Ignoring the type constant f and substitutions mapping variables to ω for simplicity, a level-0 substitution can map a variable to $2^4 - 1$ types (every non-empty subset of $\{0, 1, 2, 3\}$ represents an intersection of atoms). Since c_3 contains 4 variables there are $15^4 = 50\,625$ substitutions. The ATM that only implements the condition of*

Proposition 4.1.9 on page 61 would only instantiate four substitutions per check in line 5 for the original inhabitation question $\Gamma_4^3 \vdash ? : (0 \to 2) \cap (1 \to 3) \cap (2 \to 0) \cap (3 \to 1)$. *This would result in at least* $(15^4)^4 \approx 6.6 \times 10^{18}$ *checks in line 5. Many of the* 50 625 *possible substitutions will not help to inhabit* $(0 \to 2) \cap (1 \to 3) \cap (2 \to 0) \cap (3 \to 1)$*, though. The impact of lookahead-test used by the ATM in Figure 4.5 on page 64 to eliminate infeasible substitutions in advance is enormous: In total, the implementation of the ATM with the lookahead-test only constructed* 3889 *inhabitation questions and synthesized all solutions to the inhabitation question in approximately 1.3 seconds.*

Analysis & Discussion

The lesson learned from this experimental example is two-fold.

- First, the immense reduction in newly created inhabitation goals and execution time achieved by the implementation of the lookahead-ATM shows that any heuristic that helps to exclude substitutions early on in ATM (the earlier the better!) contains potential for drastic speedups.

  The lookahead-condition appears to be rather coarse (for example, the various numbers of arguments, k, checked for in line 12 of the ATM with the lookahead-test are not compared at all even though a *common* k must exist for all τ_i). Still, the condition is suitable to reduce the runtime by some 15 orders of magnitude in this example. We believe that the potential in restricting the number of substitutions has not been fully exploited. In particular, for many practical applications *domain-specific knowledge* may be used to exclude substitutions that are not meaningful in the practical context.

- Second, already for this small example even the ATM with the lookahead-test surpasses the resource boundaries rather quickly. The reason can be seen in the fact that the number of substitutions grows exponentially with m and n: There are $\mathcal{O}(2^{nm})$ possible substitutions. The important conclusion to be drawn here with regard to practical applications is that variable kinding is indispensable.

A further improvement can be established by restricting the ranges of substitutions of type variables. This restriction can be achieved by introducing kinded type variables. A type variable is *kinded by a set* $A \subseteq \mathbb{A}$ if it can only be instantiated by types that can be built from A (see for Section 3.2.3 on page 39 for details). Having said that, in many practical applications variable kinding comes naturally, since very often it is clear that a type variable of a

certain combinator can only meaningfully be instantiated with very few type constants. The case where every type variable may be instantiated with any type constant, as encountered in the $\mathbb{Z}_n$-example above, may therefore often prove to be far too general. Furthermore, often it does not make sense to instantiate a variable by an intersection of atoms but rather by single atoms at a time. A type variable is *atomic* if it can only be instantiated by single type constants. We enriched the implementation of the ATMs by kinded and atomic variables which was an essential step towards the applications to connector synthesis as presented in the following sections. Type variable kinding will be discussed as one optimization mechanism in the following section.

4.2 Algorithmic Optimization

Complementary to the optimizations discussed before, we now consider more general heuristical optimizations that are unspecific to type theory and that can be applied to implementations of ATMs in general. Nevertheless, we will apply type theoretic results wherever applicable in order to increase the effectiveness of the proposed optimizations.

We begin with a discussion of theoretical inhabitation algorithms and show different general optimizations to the implementation of this ATM. In the next chapter, these optimizations will be used in an implementation of relativized type inhabitation in $\mathsf{BCL}_0(\cap, \leq)$. We start by analyzing previously presented algorithms.

The theoretical algorithms in Figures 3.1 on page 44 and 4.5 on page 64 are ATMs. In order to simplify the following discussion, we mark the existential and universal choices of the Figure 3.1 on page 44 in lines 5, 7, 8, 14, and 15 in blue. With this coloring we obtain the Algorithm 4.6.

The core algorithm in Figure 4.1 on page 53 realizes the step function δ of the ATM as well as the proof-graph, which here is called execution graph. The blue part is related to the control of the ATM. An ATM distinguishes existential and universal states and we use this distinction to separate the algorithm into parts. Lines 12 and 18 contain the acceptance information `ACCEPT` and `FAIL`. The other lines (6, 10-13, 16-19) contain algorithmic information that can be regarded as a logical predicate depending on the choices before. We call this part of the algorithm *data-centric*.

After evaluating the mentioned predicate, the control of the ATM is delegated to either acceptance in lines 12 and 18 or to a transition to a universal state in line 14 generating new inhabitation questions in line 15. We call this part *control-centric*, because it only creates n new inhabitation

Algorithm 4.6: Alternating Turing Machine deciding inhabitation in $\mathsf{BCL}_k(\cap, \leq)$ with marked choices

Input: Γ, τ, k
Output: `INH1` accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$

loop:
`CHOOSE` $(x : \sigma) \in \Gamma$;
$\sigma' := \bigcap\{S(\sigma) \mid S \in \mathcal{S}_x^{(\Gamma,\tau,k)}\}$;
`CHOOSE` $n \in \{0, \ldots \|\sigma'\|\}$;
`CHOOSE` $P \subseteq \mathbb{P}_n(\sigma')$;

if $\bigcap_{\pi \in P} tgt_n(\pi) \leq \tau$ **then**
  if $n = 0$ **then**
    `ACCEPT`;
  else
    `FORALL` $(j = 1 \ldots n)$ $\tau := \bigcap_{\pi \in P} arg_j(\pi)$;
    `GOTO` *loop*;
  end if
else
  `FAIL`;
end if

questions for every argument of a chosen $(x : \sigma) \in \Gamma$. Then, the result of the inhabitation question depends on the results of these newly generated inhabitation questions.

Since we want to implement a non-deterministic ATM, we also have to apply a determinization procedure to provide a deterministic implementable algorithm. Therefore, we start with a representation for simulating the ATMs presented so far. This representation simplifies the discussion on optimizations and its data structure is closer to a deterministic implementation.

4.2.1 Execution Graph

The key idea for optimizing the Algorithm 4.1 on page 53 is to separate data and control of the ATM. Correspondingly, we will use the terms data-centric part and control-centric part of the ATM. For example, the control-centric part will include the ATM's acceptance rules for existential states in $Q_\exists$ and for universal states in $Q_\forall$. These parts will be separated and optimized differently.

We model data and control separation by a *bipartite directed graph* (BDG) distinguishing both kinds of states of $Q = Q_\forall \uplus Q_\exists$ of the ATM. The BDG represents a proof-graph. One kind of nodes, the non-deterministic choices ($Q_\exists$), are called *group nodes*, N_G, and are depicted as a rectangle. These nodes are called group nodes, because the choice in the Algorithm 4.6 on the facing page leads to n new inhabitation questions forming this group. These new questions emanate from the n chosen arguments of the inhabitation algorithm in Figure 4.6 on the preceding page. These nodes are elements of the set of universal states $Q_\forall$ of the ATM. We will associate group nodes in N_G with the choices that have been made in the ATM for the combinator x, the number of x's arguments, and an adequate subset of paths P.

The other kind of nodes are called *inhabitation nodes*, N_I, contain the inhabitation question, and are depicted as circles. These nodes are elements of the set of existential states $Q_\exists$ of the ATM.

The non-deterministic choice in state $q \in Q_\exists$ can be implemented by choosing every possible solution in q. This idea has been presented by Shapiro [1984] and is adapted to the ATM deciding inhabitation. Shapiro [1984] provides a transformation procedure of the ATM into a logical Prolog program, by coding universal states ($Q_\forall$) as logical conjunctions $\wedge$ and existential states ($Q_\exists$) as logical disjunctions $\vee$. Analogically, rules operating on execution graphs will simulate conjunctions and disjunctions of states respectively computations of ATMs.

The execution graph $\mathcal{G}_E$ is a data structure for implementing (simulating) and optimizing the ATM presented in Figure 4.1 on page 53. Its nodes consist of group nodes and inhabitation nodes. The edges between these nodes are linked to the *deterministic* simulation of the ATM.

Definition 4.2.1. ***(Execution graph)***
An execution graph $\mathcal{G}_E$ is a tuple (N_G, N_I, E) where the set N_G are the group nodes, the set N_I are the inhabitation nodes, and $E \subseteq (N_I \times N_G) \cup (N_G \times N_I)$ is the edge relation.

We call a node, n_c, *child* of a *parent* node n_p if $(n_c, n_p) \in E$ holds. In order to simplify the discussion of nodes in the execution graph, we denominate the different kinds of nodes with respect to the edge relation. The parent respectively child nodes of an inhabitation node are also called its *parent group nodes* respectively its *child group nodes*. Conversely, the parent respectively child nodes of a group node are called its *parent inhabitation nodes* respectively its *child inhabitation nodes*.

Furthermore, we define two labeling functions $\mathcal{L}_I$ and $\mathcal{L}_G$ for both kinds of nodes labeling inhabitation nodes in N_I with its inhabitation question ($\Gamma \vdash ? : \tau$) or group nodes in N_G with its choices for x, n, and P. We will

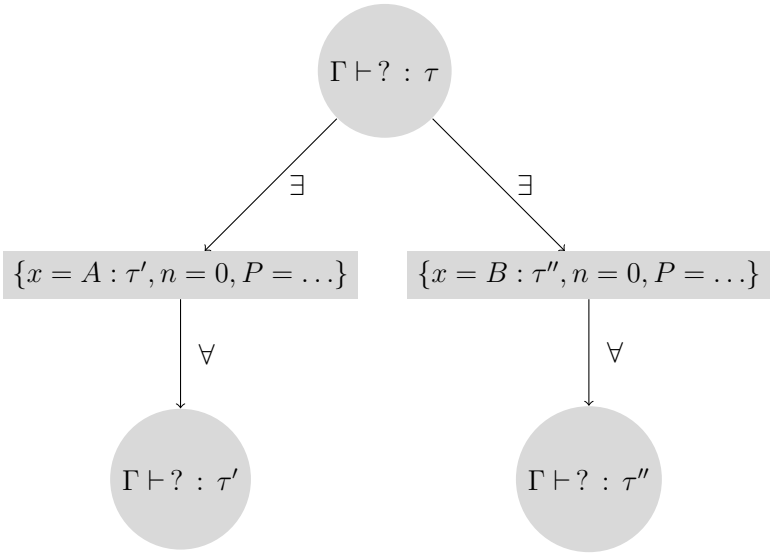


Figure 4.3: Execution Graph for $\Gamma \vdash ? : \tau$

use the convention that we omit P where the choice of P is clear from the context.

We can translate the acceptance rules of an ATM deciding inhabitation, for instance in Algorithm 4.6, to rules operating on an execution graph as follows:

1. At least one child node of a node in N_I (respectively a successor state in $Q_\exists$ accepts) must signal a successful inhabitation.

2. All children of a child group node of a node in N_G (respectively all of its successor states in $Q_\forall$ accept) must signal successful inhabitation.

3. We can evaluate these rules bottom-up.

We will use the color gray to mark an unknown state (either successful or failed) of inhabitation, green for *successful* inhabited nodes, and red for *failed* inhabitation nodes. A node is successful if its inhabitation question is inhabited otherwise the node failed. Furthermore, we will decorate edges to existential states with $\exists$ and respectively edges to universal states with $\forall$.

Figure 4.3 shows an example execution graph for the relativized inhabitation question $\Gamma \vdash ? : \tau$ with the following type environment Γ:

$$\Gamma = \{ \quad A : \tau' \to \tau, \quad B : \tau'' \to \tau, \quad C : \tau' \cap \tau'' \quad \}$$

The algorithm tries to solve this question by choosing $\{x = A : \tau' \to \tau, n = 1, P = \{\tau' \to \tau\}\}$ and $\{x = B : \tau'' \to \tau, n = 1, P = \{\tau'' \to \tau\}\}$. Both choices lead to new child inhabitation questions $\Gamma \vdash ? : \tau'$ and $\Gamma \vdash ? : \tau''$. Both inhabitation questions $\Gamma \vdash ? : \tau'$ and $\Gamma \vdash ? : \tau''$ can be directly inhabited by choosing $x = C : \tau' \cap \tau''$, $n = 0$, and $P = \{\tau'\}$ respectively $P = \{\tau''\}$.

The acceptance rules can be translated into corresponding graph markings by operational rules, wherein the state of acceptance of the ATM is translated into graph markings where the corresponding nodes N_I and N_G are marked with flags `SUCCESS` or `FAIL` representing successful inhabitation respectively failed inhabitation. Analogously to the Prolog coding by [Shapiro, 1984], acceptance rules are propagated in reverse direction of the execution graph's edges as follows (see Figure 4.4 for explanation):

Definition 4.2.2. *(**Execution graph acceptance**)*
Assume an execution graph $\mathcal{G}_E = (N_G, N_I, V)$, a parent inhabitation node $n_I^p \in N_I$ has k child group nodes $g_j \in N_G$ ($(n_I^p, g_j) \in E$ for every $j \in \{1, \ldots, k\}$), and child inhabitation nodes $n_1^j, \ldots, n_{m_j}^j \in N_I$ with $m_j \in \mathbb{N}$ of these child group nodes n_j ($(n_j, n_i^j) \in E$ for every $i \in \{1, \ldots, m_j\}$ and every $j \in \{1, \ldots, k\}$).

1. *The parent inhabitation node n_I^p is marked with*
   (a) **`SUCCESS`** *if at least one of its child group nodes $g_{j'}$ is marked with* **`SUCCESS`**.
   (b) **`FAIL`** *if all of its child group nodes $g_{j'}$ are marked with* **`FAIL`**.
2. *A group node $g_{j''}$ is marked with*
   (a) **`SUCCESS`** *if* all *its child inhabitation nodes $n_l^{j''}$ are marked with* **`SUCCESS`** *for all $l \in \{1, \ldots, m_{j''}\}$.*
   (b) **`FAIL`** *if at least one of its child inhabitation nodes $n_l^{j''}$ for all $l \in \{1, \ldots, m_{j''}\}$ is marked with* **`FAIL`**.

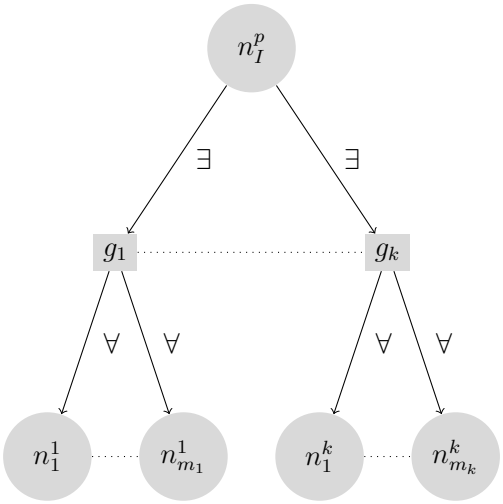


Figure 4.4: Acceptance rule in $\mathcal{G}_E$

3. An inhabitation node without any child group node is marked with

(a) **`SUCCESS`** *if the inhabitation question in this node has been inhabited directly (see Definition 3.3.2 on page 43).*

(b) **`FAIL`** *if the inhabitation question in this node cannot been inhabited directly (see Definition 3.3.2 on page 43).*

Lemma 4.2.1. *ATM in Figure 4.6 eventually accepts if and only if the root inhabitation node of the ATM's corresponding execution graph $\mathcal{G}_E$ is marked with* **`SUCCESS`**.

The proof of Lemma 4.2.1 is straightforward, because the execution graph's marking rules are designed in such a way that these marking rules simulate the eventually acceptance rules of an ATM in Section 3.4 on page 43.

Note, that cases 3a) and 3b) in the Definition 4.2.2 correspond to line 12 with the choice $n = 0$ in the Algorithm 4.6 on page 72.

The optimizations presented in the following sections can be categorized as follows:

- term level optimizations
  - normalization of intersection types
  - optimization of subtyping relation $\leq$
  - bounding the substitution in types

  - organization of type environment Γ

- prevention of redundant calculations
  - result caches
  - reusing cached results
  - cycle detection
  - restriction of result sets of inhabitants

- multi-core processing and parallelization
  - rolling processing queues
  - distributed computing barriers

These optimizations are discussed from the view point of an heuristical optimized, distributed implementation of the inhabitation algorithm. This algorithm will not only decide the inhabitation question but will also enumerate all inhabitants satisfying the inhabitation question by completely exhausting the search space of possible solutions. The various optimizations are now discussed in in more detail.

4.2.2 Term Level Optimization

Normalization of Intersection Types

A frequently executed operation is comparison and analysis of terms of intersection types. It is self-evident and necessary that a normalization of these types improves the run-time of comparisons and analyses, because normalized terms allow exploiting structural properties for these purposes. By profiling the implementation (CL)S of the inhabitation algorithm, we discovered that the normalization of intersection types consumes approximately 30% of the total computation time measured in utilized CPU usage.

Remark 4.1. *Note that we use the term* normalization *or normalized term with a different meaning than originally introduced by Hindley [1982]. Hindley's intersection type normalization uses a directed form of the distributivity rule of $\cap$ and $\to$ for a full expansion of the intersection type's term. For example, the intersection type $\tau \to \sigma \cap \sigma' \to \rho \cap \rho'$ has the normalized type $(\tau \to \sigma \to \rho) \cap (\tau \to \sigma \to \rho') \cap (\tau \to \sigma' \to \rho) \cap (\tau \to \sigma' \to \rho')$. Therefore, Hindley's normalization can lead to an exponential sized intersection type causing inefficiency when computed. Instead we use the organization of intersection types according to Lemma 3.2.3 on page 38. At this point, we are*

interested in succinct but distinct terms increasing efficiency and effectiveness of computations.

Normalization of an intersection type τ:

- removes duplicate sub-terms by exploiting idempotency of $\cap$.
- flattens intersections using associativity of $\cap$, for example $\tau = (\tau_1 \cap \tau_2) \cap (\tau_3 \cap \tau_4)$ becomes $\tau = \tau_1 \cap \tau_2 \cap \tau_3 \cap \tau_4$.
- ordering of sub-terms in τ. A hash-code over occurring sub-terms used in the ordering relation increases the effectiveness of comparison operations. After applying the order relation on an intersection type, components of τ have the following order:
  - type variables from $\mathbb{T}$,
  - type constructors,
  - $\to$-types, and
  - further intersections.

  Again, the ordering operation is applied recursively to the arguments of the sub-terms in τ.

Furthermore, this normalization guarantees that the normalized type τ has a unique hash-code, which is later used for fast comparison operations.

Optimizing the Subtyping Relation

In later chapters, subtyping is used for encoding taxonomical trees representing knowledge, for example on software connectors and software architectures. Therefore, we extended the subtyping relation $\leq$ by adding axioms containing atomic pairs of type constants to the rules presented by Barendregt et al. [1983]. Rehof and Urzyczyn [2011a] showed that deciding $\leq$ is PTIME. It follows from a result of Rehof and Mogensen [1999] that this also holds when $\leq$ is extended to a semi-lattice. Detailed information on satisfiability of other inequalities can be found in [Rehof, 1998, Pratt and Tiuryn, 1996, Tiuryn, 1992].

To speed up the subtyping decision procedure, we need to efficiently calculate the transitive closure of the atomic extension of $\leq$. Skiena [2008] lists some efficient algorithms for computing the transitive closure of relations. Coppersmith and Winograd [1987] presented an algorithm that is of $O(n^{2.3736})$ (with Stothers [2010] improvement). This algorithm has the disadvantage that

it uses matrix multiplication and therefore cannot be efficiently implemented for small relations. The algorithm by Warshall [1962] is $O(n^3)$ and comes with a succinct coding. Hence, Warshall's algorithm is used to pre-compute the transitive closure of the atomic extension of $\leq$ upfront.

Bounding the Type Substitution

In Algorithm 4.1 on page 53 in line 6 all intersections of possible substitutions are applied to σ and combined in a top-level intersection. The number of substitutions in this line in $\mathsf{BCL}_k(\cap, \leq)$ is bounded by $exp_{k+1}(p(n))$ with $p(n)$, a polynomial function in the number of type constants. Therefore, a bound on the substitutions leads to an optimization with regard to space and time resource consumption. This bound is directly controlled by the parameter k in $\mathsf{BCL}_k(\cap, \leq)$.

A straightforward optimization is to restrain substitutions only to variables occurring in a type to be substituted and restricting the substitutions by a union of the substitution ranges of each variable.

In many scenarios a variable has to be instantiated by a single type constant. For example, in a scenario for synthesizing Boolean functions as components with type constants 0 and 1, the occurring variables should only be instantiated with either 0 or 1 and not with $0 \cap 1$. The experiment conducted in Section 4.1.5 allows in a similar scenario only substitutions by type atoms. *Atomic substitutions* are an essential optimization for these usage scenarios.

This idea can be generalized by introducting kinded type variables (see for Section 3.2.3 on page 39). Here, type variables of the same kind have the same range under substitution allowing a more refined control of the used substitutions.

Later in this chapter an implementation, named (CL)S, of the decision algorithm for relativized inhabitation in $\mathsf{BCL}_0(\cap, \leq)$ is presented that allows either atomic or substitutions of level 0 for type variables.

Organization of Type Environment Γ

Inhabitation algorithms presented in Algorithms 4.1, 4.2, 4.4, and 4.5 require organized types as defined in Section 3.3 in their input. An optimization is the pre-calculation of the organized types in Γ by tacitly lifting the organization of types to a set of types like the type environment Γ as follows:

Definition 4.2.3. ***(Organized type environment)***
The organized type environment $\overline{\Gamma}$ for a type environment Γ is defined by:

$$\overline{\Gamma} = \{(x : \overline{\tau}) | (x : \tau) \in \Gamma\}.$$

This reduces the frequent and unnecessary repetition of organization of types in the relativized inhabitation algorithm for every inhabitation goal.

4.2.3 Elimination of Redundant Calculations

During the solution of an inhabitation question, new inhabitation questions are generated. Some of these newly generated inhabitation questions might occur multiple times.

Caching Results

The relativized inhabitation question $\Gamma \vdash ? : \tau$ has type environment Γ and type τ as input can either be successfully inhabited or the inhabitation fails. The result for the inhabitation question $\Gamma \vdash ? : \tau$ can be stored in one or more caches. We will say that an information or type is *cached* if the information or type is stored in a cache.

To optimize the look-up, a hash-map with τ as a key and the result of the inhabitation as value is used. We omit Γ, because it is constant for all child inhabitation questions. Therefore it is sufficient to make the look-up in the caches independent of the type environment Γ.

We will store successful inhabitations in a success cache C_S whereas failed inhabitations are stored in a fail cache C_F. We will use the short notation $\tau \in C_s$ respectively $\tau \in C_F$ if τ is stored in C_s respectively in C_F. We will use these caches to store results on child inhabitation questions and reuse these results to avoid redundant calculations. The size of the caches is bound by the space-complexity $(k+1)$--Expspace of relativized type inhabitation in $\mathsf{BCL}_k(\cap, \leq)$. However, using the optimizations presented in the previous and this chapter, the number of cache entries in C_S and C_F is linearly bounded by the number of inhabitation questions in the execution graph.

We can improve the caching of failed inhabitation questions by using Lemma 3.3.1 on page 42. Lemma 3.3.1 states that if the inhabitation of a supertype τ' of a type τ has failed, $\Gamma \nvdash ? : \tau'$, then the inhabitation of τ must also fail, $\Gamma \nvdash ? : \tau$.

By modifying the lookup method for the fail cache C_F by also comparing τ with possible τ' in C_F, we can further reduce unnecessary inhabitation requests.

Remark 4.2. *One could come up with the idea that applying a dual principle to the caching of successful inhabitants could be also an improvement. And indeed, a dual version of the Lemma 3.3.1 for τ and the success cache C_S would be correct and be an improvement for the decision problem of relativized*

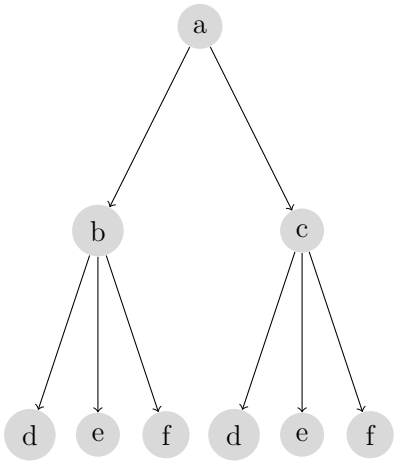


Figure 4.5: Example graph with multiple nodes

type inhabitation. However, we aim on explicitly constructing the set of all inhabitants satisfying the inhabitation question. Using the dual principle would lead to pruning the execution graph for successful inhabitation nodes and, in effect, loosing further inhabitants that are supertypes of τ, because there might exist more than one inhabitant (even subtypes) for an inhabitation goal. Then, the resulting algorithm with the dual principle would not be complete with respect to the set of inhabitants.

Execution Graph Compression

If we cache results of $\Gamma \vdash ? : \tau$, then we can reuse these results to prevent unnecessary computations for reoccurring inhabitation questions of $\Gamma \vdash ? : \tau$. This is constrained by requirement that the algorithm has to be complete. Within the execution graph the reoccuring calculation of $\Gamma \vdash ? : \tau$ can be reused from the cache by appending the subgraph of $\Gamma \vdash ? : \tau$ which has been calculated already, to the parent inhabitation node.

For the following example, we simplify the execution graph by ignoring nodes containing the information on the non-deterministic choices for x, n, and P of the algorithm.

A common compression technique for graphs is a sharing representation by using a *direct acyclic graph* (DAG). In this representation nodes that occur multiple times are eliminated and substituted by a single node that is shared under all edges in the graph. Originally this technique was proposed by Boyer and Moore [1972] for theorem-proving.[4]

[4]One year later it was included in the PhD thesis of Moore [1973].

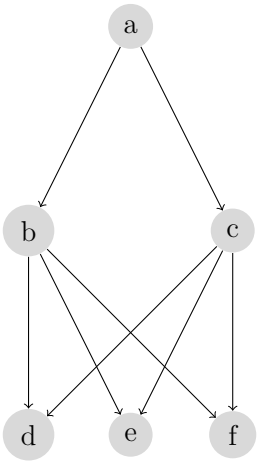


Figure 4.6: Example graph (see for Figure 4.5 on the previous page) with shared representation as DAG

Figure 4.5 on the previous page depicts an example of a graph that is compressed in a DAG for a sharing representation in Figure 4.6. This example is a special case in which only leaf nodes are shared. The technique of Boyer and Moore is of course more general.

Remark 4.3. *All paths of the graph in Figure 4.5 on the preceding page can be directly mapped to a path in graph of Figure 4.6. Therefore, this representation conserves all paths of a graph, because all paths can be reconstructed possibly causing exponential cost.*

This representation can be applied for execution graphs.

Example 4.3. *We discuss an example of a relativized inhabitation question* $\Gamma \vdash ? : \tau$ *in Figure 4.7 on the facing page. Solving this inhabitation question needs two new child inhabitation question* $\Gamma \vdash ? : \tau'$ *and* $\Gamma \vdash ? : \sigma$ *to be solved. To solve* $\Gamma \vdash ? : \tau'$*, two further inhabitation questions* $\Gamma \vdash ? : \tau''$ *and* $\Gamma \vdash ? : \tau'''$ *have to be solved as well. Without loss of generality we assume that* $\Gamma \vdash ? : \tau''$ *and* $\Gamma \vdash ? : \tau'''$ *have been solved successfully. We argue the other three cases with inhabitation results for* $\Gamma \vdash ? : \tau''$ *and* $\Gamma \vdash ? : \tau'''$ *similarly. We can cache* $\Gamma \vdash ? : \tau''$ *and* $\Gamma \vdash ? : \tau'''$ *as successful inhabitations in* C_S*, because* $\Gamma \vdash ? : \tau''$ *and* $\Gamma \vdash ? : \tau'''$ *have been solved successfully. Furthermore, it follows that* $\Gamma \vdash ? : \tau'$ *can be solved successfully. In addition, we can also cache* $\Gamma \vdash ? : \tau'$ *as successful inhabitation.*

In order to inhabit $\Gamma \vdash ? : \tau$*, we also have to solve the inhabitation question of* $\Gamma \vdash ? : \sigma$*. To solve* $\Gamma \vdash ? : \sigma$ *we again have to solve the inhabitation question for* $\Gamma \vdash ? : \tau'$ *in Figure 4.8 on the next page.*

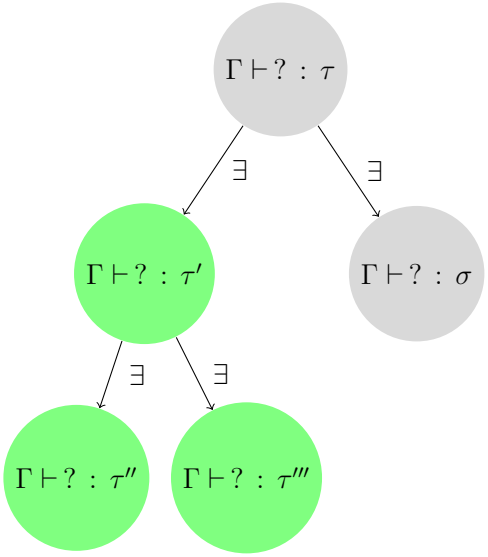


Figure 4.7: Execution Graph with cached result for $\Gamma \vdash ? : \tau'$

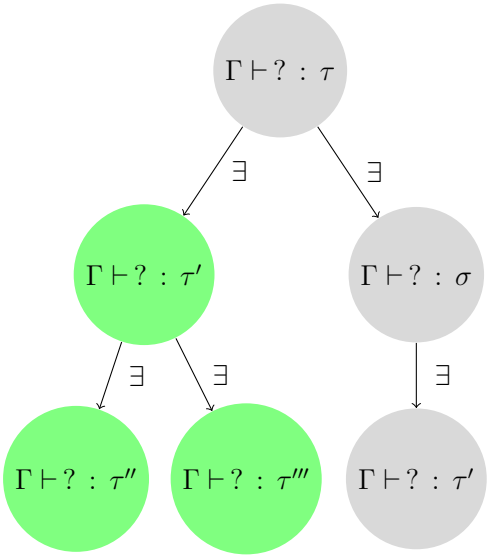


Figure 4.8: Execution Graph with a reoccuring $\Gamma \vdash ? : \tau'$ node

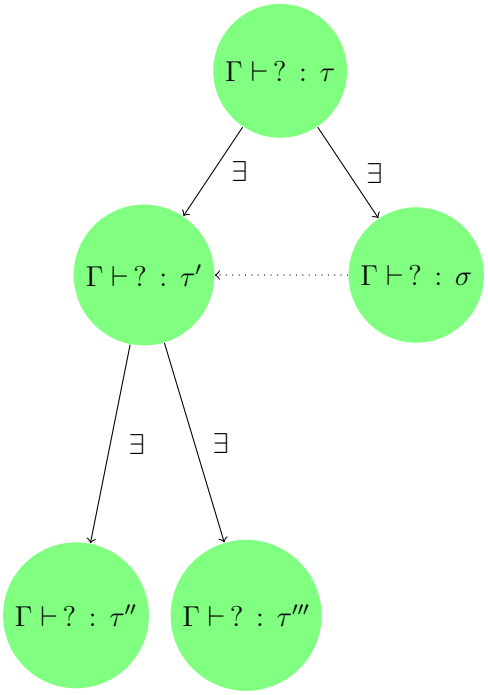


Figure 4.9: Execution Graph with a reused $\Gamma \vdash ? : \tau'$ node

Since we have $\Gamma \vdash ? : \tau'$, $\Gamma \vdash ? : \tau''$, *and* $\Gamma \vdash ? : \tau'''$ *cached. We can use the cached result for* $\Gamma \vdash ? : \tau'$ *and prevent a recalculation of* $\Gamma \vdash ? : \tau'$. *These results are depicted in Figure 4.9.*

After reusing the cached result of $\Gamma \vdash ? : \tau'$, *we can conclude that* $\Gamma \vdash ? : \sigma$ *can be inhabited successfully. Therefore, the initial inhabitation question* $\Gamma \vdash ? : \tau$ *can be inhabited successfully.*

The technique of Boyer and Moore is here used to *prevent* the creation of duplicate nodes. This is achieved by maintaining a hash-map of existing nodes. A lookup in this hash-map is used, to determine if a new node has to be created or an existing is shared.

We have also to extend this rule for operating on group nodes for inhabitation questions of arguments of a chosen x in an execution graph.

Cycle Detection

The set of inhabitants in $\mathsf{BCL}_k(\cap, \leq)$ can be of infinite cardinality. Rehof and Urzyczyn [2011a] proved that in the case of infinite many inhabitants cyclic structures *must* occur in the computation of the inhabitation algorithm. We will demonstrate this fact with a small example.

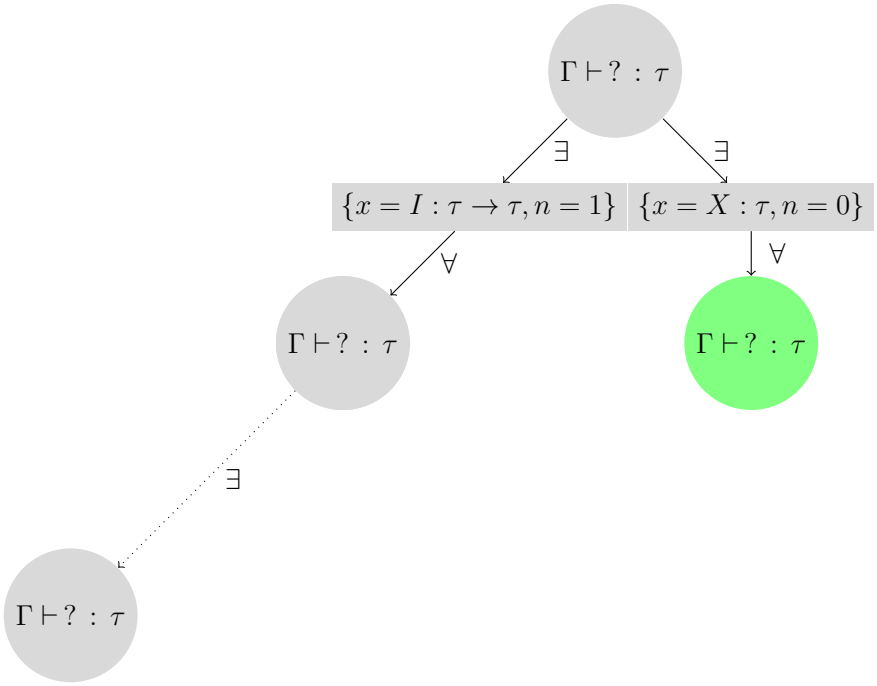


Figure 4.10: Execution Graph containing a cyclic solution

Example 4.4. *For example, we have a type environment* Γ *consisting of two combinators:*

$$\Gamma = \{ \quad I : \tau \to \tau, \quad X : \tau \quad \}$$

I is the identity function $\tau \to \tau$ and x is a combinator of type τ. The inhabitation question $\Gamma \vdash ? : \tau$has the inhabitants X, IX, $I(IX), I(I(IX)), \ldots$. In general the inhabitant can be written as a regular expression $(I)^*X$, with I^* noting a sequence of zero, one, or many occurrences of I. Note, that this does not mean that there exists an infinite application of I. The applicative terms constructed by the regular expression are still finite terms. The execution graph is depicted in Figure 4.10.

An algorithm without cycle detection would also lead to an infinite computation. We use a cycle detection algorithm, which traverses the execution graph from the actual position, where a new child inhabitation question $\Gamma \vdash ? : \sigma$ is posed, up to the root inhabitation question. We compare each

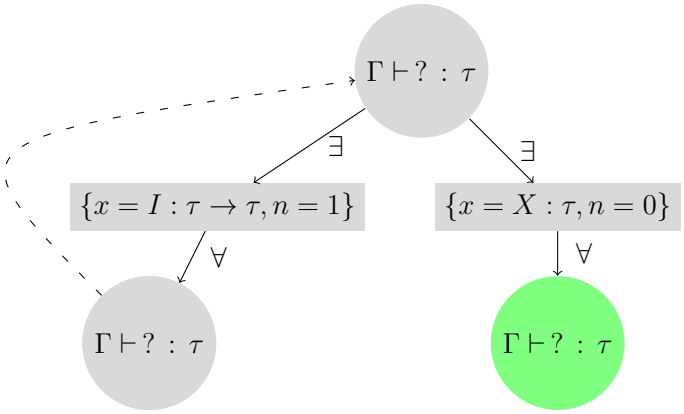


Figure 4.11: Execution Graph containing a cyclic solution with a backward edge

node ($\Gamma \vdash ? : \sigma'$) on the path from the new child inhabitation question $\Gamma \vdash ? : \sigma$ to the root inhabitation question and if $\sigma' \leq \sigma$ holds, then we can use the solution of σ to satisfy the inhabitation question of $\Gamma \vdash ? : \sigma$. In this case we can use a backward edge from this position to the one of $\Gamma \vdash ? : \sigma'$. For the example above, this is shown in Figure 4.11.

Remark 4.4. *Note that the node in the left-corner cannot be removed since we would loose the information that I can be used.*

Since we can inhabit $\Gamma \vdash ? : \tau$ by choosing $\{x = X : \tau, n = 0\}$, we can mark the cyclic solution in the left part of Figure 4.11 as successful, too. We can do this, because the left hand side inhabitation question is a subtype (in this case identical) with the successfully inhabited right hand side. This leads to Figure 4.12 on the facing page and the expected successful inhabitation of the root inhabitation question $\Gamma \vdash ? : \tau$. Also note that this operation is not possible in an ATM. An ATM would have to roll out the complete computation contained in a cyclic path. Using an inductive argument over the path length, it is easy to see that the acceptance in this case can be simulated by the compressed cyclic graph.

Note 4.1. *Cyclic structures in the execution graph denoting inhabitants are representable as a term by using a regular tree grammar.*

Restricting the Result Set of Inhabitants

In some usage scenarios there is no need to return the complete set of all inhabitants of an inhabitation request. Either only one solution is needed or a quick solution is favored against time-intensive finding of all inhabitants.

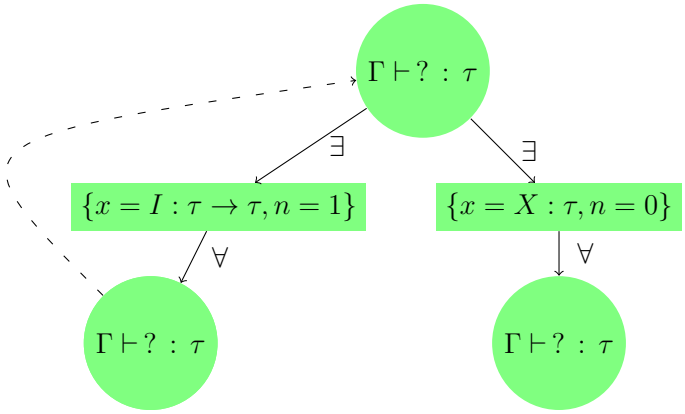


Figure 4.12: Execution Graph marked with success containing a cyclic solution with a backward edge

In case that finding single solution for inhabitations is favored, a unique specification property stating that a type environment Γ can always be rewritten suitably for an inhabitation question such that this inhabitation question has a unique inhabitant.

Proposition 4.2.2 (Unique specification [Rehof and Urzyczyn, 2011a,b])**.** *For every combinatory expression e there exists an environment Γ_e and a type τ_e such that e is the unique term with $\Gamma_e \vdash e : \tau_e$.*

The proof for this property by Rehof and Urzyczyn [2011a,b] can easily be transferred to $\mathsf{BCL}_k(\cap, \leq)$. On the contrary, if an inhabitation question returns too many inhabitants, then the type environment Γ might be underspecified. Either such an underspecification is intended or the type environment Γ needs a more refined specification.

In the case of a single or fixed number of solutions, counters for the number of inhabitants in the algorithm can be used to terminate the computation whenever sufficient number of inhabitants have been found.

4.2.4 Parallel Computation

The execution graph is evaluated based on its edge-relation. Every inhabitation question of the form $\Gamma \vdash ? : \tau$ can be inhabited separately. The inhabitation algorithm is encapsulated in the nodes. The inhabitation algorithm does not modify these nodes. Such problems are called data-parallel because the non-deterministic choices are the same in all nodes. Only a dependency to the parent node must be respected. This fact does not hinder scalable parallelism. The first solution could be a data-parallel algorithm

using barriers on each level. At this point, the overall problem is to find an optimal task-schedule for maximizing the CPU-utilization (indirectly measured by thread utilization).

Different strategies exist for traversing a graph like an execution graph. A *depth-first search* (DFS) would traverse into the depth of an execution graph. This strategy would reduce the advantage of reusing results. This can easily be seen in an execution graph with varying path depths. In DFS traversal, this short path is evaluated after the deep path. Therefore, the cached result will not be available during the traversal of the deep path.

This disadvantage for DFS points to the idea of one other traversal strategy. The *breadth-first search* (BFS) traverses all nodes with same distance (path length in the execution graph) or in trees called level at the same time. BFS can be implemented with a (thread-safe) queue operating in FIFO order. In our case, the inhabitation questions in the queue are linked to corresponding $Q_\exists$ nodes in the execution graph.

If two or more tasks are solving the same inhabitation question, we terminate the remaining tasks after one task finds a solution in order to minimize unnecessary computations, for instance, by using cancellation tokens.

The implemented algorithm in (CL)S uses a thread-pool whose size depends on the number of cores of a given computer. We have conducted experiments that shown that either $2n$ or only n threads should be generated for a n-core CPU to be optimal. There exist two causes for the sizes $2n$ and n. The first cause is that the inhabitation task is very computationally intensive and therefore a one to one correlation of tasks to cores is plausible. The second cause is that the costs for interprocess communication, thread context-switching, and thread blocking are increasing by the number of tasks and these costs dominate the utilization for higher figures greater than $2n$ threads for n CPU cores.

(CL)S offers two possible parallelization strategies:

- One strategy is called *rolling queue strategy* (RQS), which uses the level in an execution graph as a barrier. This strategy is also known as barriers in distributed computing. A barrier stops tasks that reach the barrier until all tasks reach this barrier. Then the barrier resumes all tasks. Barriers are placed at the same depth in the execution graph and can be implemented by rolling queues. The depth is measured in the number of computation steps from the root. The definition of depth directly corresponds to the common definition of a level in a tree.

  Example 4.5. *For example, we assume a thread-pool that contains four threads. And we have an execution graph which has the following sequence of number of nodes at each level* $(1, 3, 5, 9)$*. Then we have the fol-*

lowing number of utilized threads working concurrently $(1, 3, 4, 1, 4, 4, 1)$ *with length* 7*. This schedule is sub-optimal with respect to the optimal solution* $(1, 3, 4, 4, 4, 2)$ *with length* 6 *obeying the dependencies, because the optimal solution needed one step less than the previous schedule.*

- The second strategy is more flexible but more complex to implement. In this strategy, generic task partitioners are used to partition a given set of tasks and assign these partitions to threads for execution. A partitioner called *term complexity partitioner* (TCP) has been implemented and used for experiments described later. TCP uses the term complexity of τ of an inhabitation question $\Gamma \vdash ? : \tau$ to cluster sets of inhabitation questions. Here, the term complexity is measured in the size $\|\tau\|$ which is the number of nodes in the abstract syntax tree of τ.

  Optionally, another partition strategy could be to assign unique threads to types containing variables and using the rest of the cores shared for types only containing type constants. This can be useful, because we know from the analysis of the theoretical algorithm that computing the substitutions is very resource-consuming in memory and CPU-time.

  A good survey on partitioning algorithms can be found in the work by Chamberlain [1998]. Walshaw et al. [1997] are presenting notable experimental results for an algorithmically similar problem called unstructured meshes. Partitioners are an ongoing research topic in *high performance computing* (HPC).

Both strategies allow a scattering algorithm to distribute the computation input among workers equally for optimal load balancing.

Experiments on these parallelization strategies have been conducted with the (CL)S implementation applied to various example inhabitation questions on a quad-core (4), on a hex-core (16), and on a 64-core computer. These experiments have been conducted on two different systems.

- A PC was used for conducting the experiments on a quad-core computer. Its specification is included in the Appendix in Section B.3.1 on page 209.

- A compute cluster server was used for conducting the experiments on a 16 and 64 core computer. Its specification is included in the Appendix in Section B.3.2 on page 209.

More detailed information on the systems and the measurement method can be found in the Appendix B.

For analysis, the performance workload has been logged. This log shows the *utilization ratio* (UR) $\overline{U_R}$ of the aggregated normalized sum of utilized

threads. With a thread-pool size T and a number of maximal computation steps N of an execution graph, and a number of utilized threads T_i in computation step $i \in \{1, \ldots, N\}$, UR can be written as:

$$\overline{U_R} = \frac{\sum_{i=1}^{N} \frac{T_i}{T}}{N}$$

$\overline{U_R}$ for Example 4.5 above $(1, 3, 5, 9)$ with utilization $(1, 3, 4, 1, 4, 4, 1)$ is 0.643. In our setting with a single root node, $\overline{U_R}$ is always less than 1. Since every inhabitation question has to try every $x \in \Gamma$, different numbers of arguments n, and paths P the measured computation time in a node N_I is nearly constant. The variance of the computation time is caused by the optimization strategies in the computation of substitutions. Multi-tasking also inflicts additional fluctuations in the measurements. The variance as measured so far is insignificant. Therefore, $\frac{T_U^i}{T}$ corresponds (by multiplication of a constant factor) to the classical utilization definition with utilization of processor P_i as

$$Utilization(P_i) = \frac{ComputeTime(P_i)}{IdleTime(P_i) + ComputeTime(P_i)}.$$

Here, an advantage of using $\overline{U_R}$'s definition is to be a relative figure delivering a dimensionless quantity.

Table 4.2 and Plot 4.13 show the experimentally measured $\overline{U_R}$ for the *rolling queue strategy* (RQS) for different inhabitation problems. The details to the different inhabitation problems are presented in the Appendix in Section B.2 on page 207. The measured speedup for RQS is shown in Table 4.3 on page 92.

Problem	$\overline{U_R}$		
	4 cores	16 cores	64 cores
Problem 1 / Tracking (cf. pp. 7)	0.831	0.781	0.697
Problem 2 / Γ_4^2 (cf. pp. 66)	0.911	0.867	0.828
Problem 3 / Γ_4^3 (cf. pp. 66)	0.937	0.178	0.43
Problem 4 / Γ_3^4 (cf. pp. 66)	0.856	0.742	0.538
Arithmetical mean	0.88	0.777	0.623
Standard deviation	0.048	0.057	0.151

Table 4.2: Experimentally measured $\overline{U_R}$ for RQS

Table 4.4 on page 92 and Plot 4.14 show the same results for experimentally measured $\overline{U_R}$ for the *term complexity partitioner* (TCP). The measured speedup for TCP is shown in Table 4.5 on page 92.

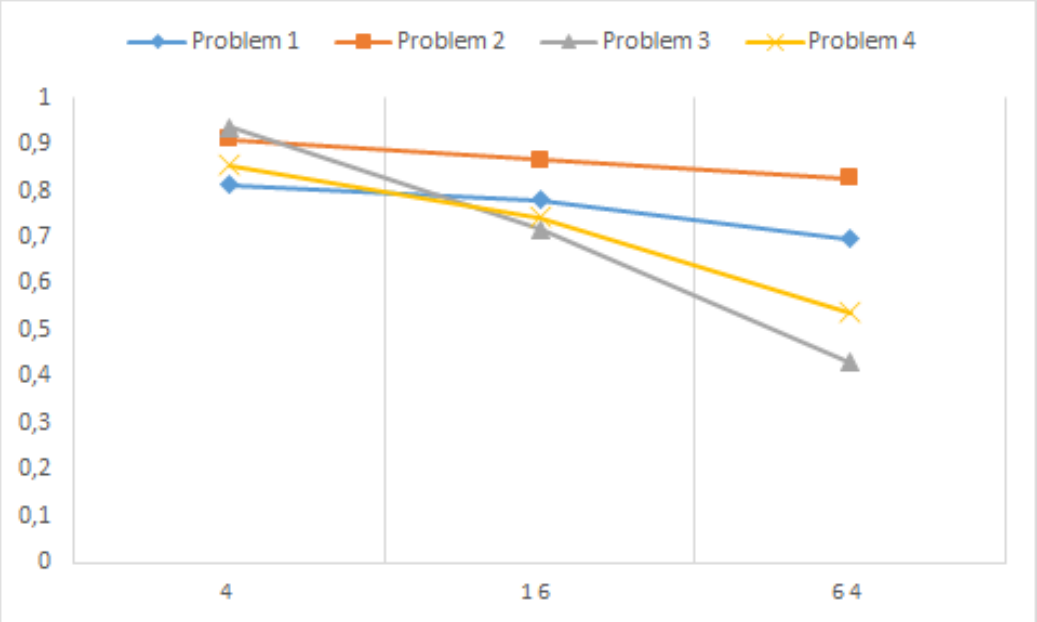


Figure 4.13: Plot of Experimentally measured $\overline{U_R}$ for RQS

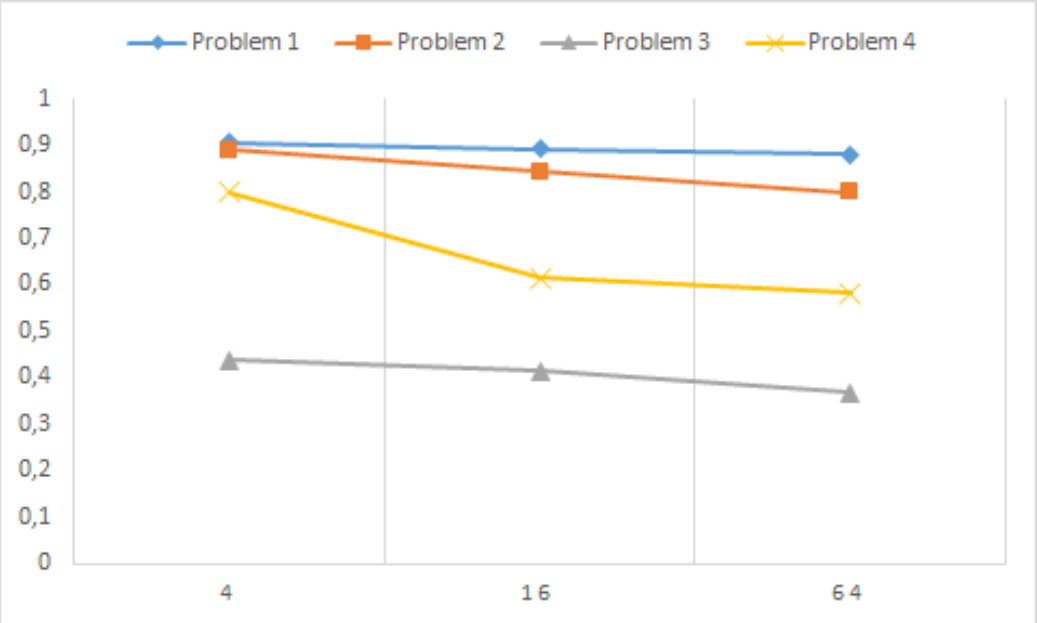


Figure 4.14: Plot of Experimentally measured $\overline{U_R}$ for TCP

Problem	Speedup RQS		
	4 cores	16 cores	64 cores
Problem 1 / Tracking (cf. pp. 7)	3.252	12.496	44.608
Problem 2 / Γ_4^2 (cf. pp. 66)	3.644	13.872	52.992
Problem 3 / Γ_4^3 (cf. pp. 66)	3.748	11.488	27.52
Problem 4 / Γ_3^4 (cf. pp. 66)	3.424	11.872	34.432
Arithmetical mean	3.517	12.432	39.888
Standard deviation	0.193	0.906	9.705

Table 4.3: Experimentally measured speedup for RQS

Problem	$\overline{U_R}$		
	4 cores	16 cores	64 cores
Problem 1 / Tracking (cf. pp. 7)	0.907	0.893	0.881
Problem 2 / Γ_4^2 (cf. pp. 66)	0.89	0.943	0.798
Problem 3 / Γ_4^3 (cf. pp. 66)	0.438	0.414	0.367
Problem 4 / Γ_3^4 (cf. pp. 66)	0.8	0.614	0.581
Arithmetical mean	0.759	0.691	0.657
Standard deviation	0.19	0.19	0.2

Table 4.4: Experimentally measured $\overline{U_R}$ for TCP

Problem	Speedup TCP		
	4 cores	16 cores	64 cores
Problem 1 / Tracking (cf. pp. 7)	3.628	14.288	56.384
Problem 2 / Γ_4^2 (cf. pp. 66)	3.56	13.488	51.072
Problem 3 / Γ_4^3 (cf. pp. 66)	1.752	6.624	23.488
Problem 4 / Γ_3^4 (cf. pp. 66)	3.2	9.824	37.184
Arithmetical mean	3.035	11.056	42.032
Standard deviation	0.758	3.063	12.797

Table 4.5: Experimentally measured speedup for TCP

Both data sets have been measured with a sample size of 50 runs.

In Tables 4.2 and 4.4, the case of experimentation with one single CPU core, that is identical to a sequential processing of the inhabitation question, is omitted, because the utilization of this case will be constant, $\overline{U_R} = 1$.

The results of the measurements in Tables 4.2 and 4.4 are discussed now.

1. In Table 4.2 and also in Table 4.4, the figures of the utilization $\overline{U_R}$ are decreasing as we increase the number of CPU cores assigned to the inhabitation problem. This can be explained by idle threads waiting for new inhabitation requests in the working queue to process.

2. The results suggest that the TCP strategy has a better utilization $\overline{U_R}$ for multi-core systems (64 cores), whereas RQS's deviation is smaller.

3. The smaller mean values of the deviation of RQS of $\overline{0.085}$ compared to $\overline{0.193}$ of TCP show that RQS is more robust than TCP with respect to varying problems. That makes RQS a more suitable candidate as a robust default algorithm for many- and multi-core settings. Whereas TCP has more potential for optimization because of its genericity.

These results suggest to use RQS as default algorithm for the implementation of (CL)S and to use TCP for a future optimization strategy.

Alternative parallelization approaches

A second parallelization approach is thinkable. This method would parallelize the data-centric part of the algorithm. Both parallelization strategies exclude each other, because we have a finite set of cores for executing the tasks. In experiments, only optimizing the data-centric part turned out to be not that efficient. The reason is that within the data-centric part many computations are interconnected and dependent.

This approach goes well with embarassingly parallel problems. After analyzing experiments, the Utilization Ratio of this approach was significantly lower (≈30%) than the parallelization of the control-centric part. This does not mean that there is no way to parallelize the data-centric part. It rather means that this approach is more complicated and needs a sophisticated analysis and a more advanced algorithm design.

A third approach is a hybrid of data-centric and control-centric parallelization. This approach needs a more fine grained control on the task to core assignment to optimize the utilization ratio of the computation. This approach has not been pursued yet but seems to be a promising idea for a future optimization especially for multi-core systems.

Nevertheless, the heuristical optimizations discussed before lead to an implementation that is very effective for real-world problems and has been used in many different scenarios.[5]

[5]These optimizations have been used for synthesis problems for Lego® NXT Mindstorm Robots embedded controller code synthesis, GUI-Synthesis, Cloud configuration synthesis, and Workflow-Synthesis.

4.3 A Distributed Algorithm for $\mathsf{BCL}_0(\cap, \leq)$ Inhabitation

The distributed inhabitation algorithm for $\mathsf{BCL}_0(\cap, \leq)$ is implemented separately for:

1. a data-centric part
2. a control-centric part

as advocated in Section 4.2.1 on page 72. The algorithm includes the heuristical optimizations listed in Sections 4.1 to 4.2 on pages 52–71. The data-centric part of the algorithm is called `InhabOptimized` and depicted in Figure 4.7 on page 96.

The algorithm is a deterministic implementation of the algorithm in Figure 4.5 on page 64 with a modification that the recursion of the ATM is moved to a separate, central control algorithm. This control algorithm will be presented in two different versions. The first control algorithm will be sequential and easier to discuss. The second control algorithm will be based on the sequential algorithm but be concurrent.

Data Algorithm (Inhabitation)

The algorithm presented in Figure 4.5 on page 64 is using an ATM that contains non-deterministic choices, which is a theoretical but not a practical construct. Therefore, we have to simulate the occurring non-deterministic choices by a deterministic construct. *deterministic Turing machines* (DTMs) can simulate *non deterministic Turing machines* (NTMs) [Papadimitriou, 1994]. This simulation uses a DTM whose configurations represent multiple configurations of the NTM. Now the step operation of the DTM is to visit each of the NTM's successor configurations in turn, executing a single step at each visit, and spawning new configurations whenever the transition relation defines multiple continuations.

We can now use a similar approach to simulate the non-deterministic choices in the ATM by deterministic choices. The non-deterministic choices of the form `CHOOSE` $q \in Q$ with predicate $q \in Q$ are replaced by `FORALL` $q \in Q$ to probe all possible valuations of q. The similarities to the optimized algorithm in Figure 4.5 are obvious.

We will discuss the changes that are related to the algorithmic optimization discussed in Section 4.2 on page 71:

- *Line 2*: A Boolean mutable variable `PossibleSuccess` is initialized with false. This variable is used to determine, if any new inhabitation

question or any direct inhabitation (with no arguments $n = 0$) occurred in this node. If this is not the case, node τ in the execution graph can directly be marked with *FAILED* in line 35. If during the processing of this node either a direct inhabitation is successful or new inhabitation questions have been generated in line 18 respectively line 23, then the variable `PossibleSuccess` is set to *TRUE*.

- *Line 14*: At this line a path satisfying the condition $\pi_i \leq \tau$ has been found. In this case a new group node labeled with the choices for $x \in \Gamma$, n, and π is created as child of the current node τ.

- *Line 17*: The choice of $x \in \Gamma$ must have been a combinator $(x : \sigma)$ where σ is a type without any arguments since $n = 0$ in this case. The current node τ can be marked as successful and the successful inhabitation of τ with x can be propagated in the execution graph.

- *Line 20*: In this case we have found $(x : \sigma)$ with a possibly successful path satisfying $\pi_i \leq \tau$ but with arguments that must be inhabited. Therefore, we cannot mark this node τ neither as *SUCCESS* nor *FAILED*. Instead we mark the current node τ as *UNKNOWN*.

  We will later see in the control algorithm that nodes marked with *UNKNOWN* at the end of the complete inhabitation processing are marked with *FAILED*.

An excerpt of the concrete implementation in F# of Algorithm 4.7 is shown in a source code listing in Appendix A.1 on page 205.

Sequential Control Algorithm

The control-part is shown in Algorithm 4.8 on page 97. This algorithm calls `InhabOptimized` in line 22. Either `InhabOptimized` returns (see for Figure 4.7)

- *SUCCESS* in the case that new inhabitation nodes containing new inhabitation questions have been created or

- *FAILED* in the case that the node has been marked as failed and no new inhabitation nodes have been created.

The control algorithm simulates the transition Δ and acceptance rules of an ATM presented in Section 3.4 on page 43 and by Chandra et al. [1981].

We will discuss the sequential version of the algorithm for clarity and will refer to the differences for a concurrent version of Algorithm 32 as needed.

Algorithm 4.7: Algorithm `InhabOptimized` for (CL)S

Require: Γ, $\tau = \bigcap_{i \in I} \tau_i$ are organized

Input: Γ, τ
`PossibleSuccess`:=**false**
for all $(x : \sigma) \in \Gamma$ **do**
  write $\sigma \equiv \bigcap_{j \in J} \sigma_j$
  for all $i \in I, j \in J, m \leq \|\sigma\|$ **do**
    candidates(i, j, m):=`Match`$(tgt_m(\sigma_j) \leq \tau_i)$
  end for
  M:=$\{m \leq \|\sigma\| \mid \forall_{i \in I} \exists_{j \in J} :$ `candidates`$(i, j, m) =$ **true**$\}$
  if M$\neq \emptyset$ **then**
    for all $m \in M$ **do**
      for all $j_i \in J$ with `candidates`$(i, j_i, m) =$ **true do**
        for all S_i is a substitution **do**
          for all $\pi_i \in \mathbb{P}_m(\overline{S_i(\sigma_{j_i})})$ with $tgt_m(\pi_i) \leq \tau_i$ and
          $\forall 1 \leq l \leq m\ \forall \pi' \in \overline{arg_l(\pi_i)}\ \exists (y : \rho) \in \Gamma\ \exists$ a path ρ' in ρ
          $\exists k$: `Match`$(tgt_k(\rho') \leq \pi') =$ **true do**
            Add group node to τ
            if $n = 0$ **then**
              Mark τ with **true**
              Propagate result τ in execution graph $\mathcal{G}_E$
              `PossibleSuccess`:=**true**
            else
              Mark τ with *UNKNOWN*
              for all $l \in \{1 \dots m\}$ **do**
                Add inhabitation node $\bigcap_{i \in I} arg_l(\pi_i)$ to g
                `PossibleSuccess`:=**true**
              end for
            end if
          end for
        end for
      end for
    end for
  end if
end for
if `PossibleSuccess` **then**
  return *SUCCESS*
else
  Mark τ as *FAILED*
  return *FAILED*
end if

Algorithm 4.8: Sequential Control Algorithm for (CL)S

```
1: Input: Γ, τ
2: Q := Initialize Working Queue
3: C_S := Initialize Success Cache
4: C_F := Initialize Fail Cache
5: Γ̄ := Organize Γ
6: τ_N := Normalize τ
7: root := Create inhabitation node τ_N
8: Enqueue root in Q
9: while Q has elements do
10:   q := Dequeue from Q
11:   if q is part of cycle then
12:     Mark q as end point of cycle
13:   end if
14:   τ̄' := Organize τ' in q
15:   if τ̄' ∈ C_S then
16:     Link parent τ̄' to existing τ̄' ∈ C_S
17:   else
18:     if ∃σ ∈ C_F.τ̄' ≤ σ then
19:       Mark τ̄' as FAILED
20:     end if
21:     if τ̄' ∉ C_F then
22:       if InhabOptimized(Γ̄,τ̄')=FAILED then
23:         Add τ̄' to C_F
24:         Mark τ̄' as FAILED
25:       end if
26:       Propagate result of τ̄' in execution graph G_E
27:     end if
28:   end if
29: end while
30: Fix cycles recursively in root
31: Mark UNKNOWN nodes as FAILED and propagate result
32: if root is marked with SUCCESS then
33:   return ACCEPT
34: else
35:   return FAIL
36: end if
```

Thus, we will postpone a discussion about synchronization and threading issues to Section 4.3 in favor of clarity.

- *Line 2 to line 7:* In these lines the data structures used by the control algorithm are constructed and initialized:

  1. The working queue `Q` is initialized as empty queue. This working queue is used for the BFS traversal through the execution graph.
  2. The success cache C_S is a hash map from integers to nodes mapping hash numbers of types to their nodes in the execution graph. Types/nodes in this cache have been successfully inhabited by the algorithm so far. This structure will be used to compress the execution graph as described in Section 4.2.3 on page 81 and Section 4.2.3 on page 80.
  3. The fail cache C_F is a hash map from integers to nodes mapping hash numbers of types to their nodes in the execution graph. Types/nodes in this cache have failed to be successfully inhabited by the algorithm so far. This structure will be used to stop further inhabitation attempts that have failed before. This implements the optimization discussed in Section 4.2.3 on page 80.
  4. The type environment Γ is organized as described in Section 4.2.2 on page 79 to prevent repetitive organization.
  5. The goal type τ of the inhabitation question $\Gamma \vdash ? : \tau$ is normalized to τ_N as described in Section 4.2.2 on page 77.
  6. The optimization of the atomic extension of the subtyping relation $\leq$ as discussed in Section 4.2.2 on page 78 is omitted for brevity.

- *Line 7 and line 8*: The root node of the execution graph is created with τ_N and also placed in the working queue as primary inhabitation question.

- *Line 9 to line 29*: The **while**-loop iterates over the elements in the working queue `Q` for traversing the execution graph in BFS.

- *Line 10*: The current inhabitation question with goal q is retrieved from the working queue Q.

- *Line 11*: This line checks if node q is part of a cycle. The cycle detection is described in Section 4.2.3 on page 84.

- *Line 12*: If node q is part of a cycle with a starting point of the cycle q' a separate cycle edge from q to q' is created. This step is also described in Section 4.2.3 on page 84.

- *Line 14*: The type in node q is organized and assigned to τ' because the inhabitation algorithm requires organized input.

- *Line 15*: If the organized type $\overline{\tau'}$ creates a cache hit, $\overline{\tau'}$ must have been inhabited successfully before. We can omit its computation and reuse the former result of the inhabitation of $\overline{\tau'}$. This is described in Section 4.2.3 on page 80 in more detail. A direct look-up method can be used. Remark 4.2 on page 80 provides further information on this decision.

- *Line 16*: We reuse the previous result of the inhabitation of $\overline{\tau'}$ by relinking the result's subgraph of the execution graph beneath the former inhabitation to q's parent node.

- *Line 18*: If there exists a supertype σ of $\overline{\tau'}$ in the fail cache C_F, then the inhabitation of $\overline{\tau'}$ will also fail. This follows because $\overline{\tau'} \leq \sigma$ and the inhabitation of the supertype σ of $\overline{\tau'}$ has failed before and we can omit this computation. This is described in Section 4.2.3 on page 80 in addition with Lemma 3.3.1 on page 42.

- *Line 19*: The inhabitation of $\overline{\tau'}$ must have failed before and we can mark q with *FAILED*, accordingly.

- *Line 21*: The current inhabitation goal $\overline{\tau'}$ is neither cached in C_S nor in C_F. It means, we have no information on $\overline{\tau'}$.

- *Line 22*: We must call the inhabitation procedure `InhabOptimized` of the Algorithm 4.7 on page 96.

- *Line 23*: The inhabitation of $\overline{\tau'}$ did not produce a direct inhabitation and no additional inhabitation questions have been created. There is no possibility for a successful inhabitation of $\overline{\tau'}$ and we can add $\overline{\tau'}$ to the fail cache C_F to prevent further inhabitation attempts.

- *Line 24*: $\overline{\tau'}$ can never be inhabited. Accordingly, the node q of $\overline{\tau'}$ is marked with *FAILED*.

- *Line 26*: For any possible outcome of the inhabitation question for $\overline{\tau'}$, the result must be propagated in the execution graph to set nodes either to *SUCCESS* or *FAIL* or leave them unchanged. This propagation is

described in Section 4.2.1 on page 72 and the **while**-loop can proceed processing the next element in the working queue.

- *Line 30*: After processing all elements, cycles must be set to the states (*SUCCESS* or *FAIL*) according to the cycle begin's state. If the cycle has been inhabited by another branch of the execution graph, the cycle itself is inhabited using finitely many cycles together with at least one successful branch.

- *Line 31*: If there are any nodes in the execution graph marked with *UNKNOWN*, then mark these nodes with failed.

- *Line 32 to line 36*: If the root inhabitation question is marked with *SUCCESS*, $\Gamma \vdash ? : \tau$ is inhabited. The result is returned by the control algorithm.

Remark 4.5. *Note that Algorithms 4.7 and 4.8 are deterministic algorithms and* not *ATMs anymore. The procedure call in line 22 of Algorithm 4.8 would be invalid within an ATM.*

Concurrent Control Algorithm

The concurrent algorithm (Algorithm 4.9) is based on the sequential algorithm in the previous subsection. The **while**-loop is now executed by separate worker threads in a master-slave pattern [Buschmann et al., 1996, pages 321ff]. These threads are spawned by a main thread, which waits for all worker threads to be stopped.

The working threads are stopping their execution upon a termination signal that is triggered whenever a semaphore S reaches zero. We initialize the semaphore at the start of the threads in line 12 with the number of working threads in the thread-pool. After finishing one inhabitation question the semaphore is decreased in line 32.

If the number reaches zero an extra watchdog thread monitoring the semaphore raises a termination signal for the working threads. The working threads receive the termination signal and stop their execution. The main thread is waiting in line 36 for the termination of all worker threads. After all worker threads stopped their execution, the main thread continues analogously to the sequential algorithm.

This procedure is needed, because when processing of the inhabitation questions concurrently some worker threads might become idle if not enough inhabitation questions are contained as tasks in the working queue. The

construction, initialization, and destruction is very resource consuming. Therefore, it would be a waste of resource to terminate these threads and restart them with a new task. Hence, these working threads should terminate if and only if no inhabitation questions are in the working queue *and* no new inhabitation questions will be generated.

The working queue is implemented as work-stealing queue that has been presented by Blumofe [1994] and later by Blumofe and Leiserson [1999].

We will discuss the concurrent algorithm in more detail:

- *Line 2 to line 31:* This part is similar to the sequential control algorithm except that the data structures are thread-safe using thread-safe caches, a shared memory[6] structure for execution graph with locks, a semaphore, and a signal for the termination of threads, and a thread-safe queue.

- *Line 9:* Spawns n concurrent working threads coming from a thread-pool that are executing the task defined in line 11 to line 33.

- *Line 11:* This line corresponds to the one of the sequential algorithm. Also a check for the presence of a set signal is used to inform the thread to abort its task. If such a signal is received, then the **while**-loop is exited.

- *Line 12:* In this line the semaphore S is incremented by one because the thread has an inhabitation question of q as task to process.

- *Line 32:* In this line the semaphore S is decreased by one because the thread has finished an inhabitation question q as task to process.

- *Line 13 to line 31:* These lines correspond to the lines in the sequential algorithm except that concurrency must be treated by thread-safe data-structures as described above.

- *Line 36:* The main thread waits for all n spawned worker threads to finish by putting the main thread to sleep. It is woken up automatically by a signal that all worker threads have finished their tasks.

- *Line 37 to line 43:* This part is similar to the sequential control algorithm because these lines are independent of concurrency. This part belongs to the main thread and is therefore sequential.

An implementation of an execution graph as *convergent or commutative replicated data type* (CRDT) graph, e.g. discussed by Shapiro et al. [2011] and

[6] In the sense of a shared repository pattern in [Buschmann et al., 1996, pages 202ff].

Algorithm 4.9: Concurrent Control Algorithm for (CL)S

```
1: Input: Γ, τ
2: Q := Initialize Working Queue
3: C_S := Initialize Success Cache
4: C_F := Initialize Fail Cache
5: Γ̄ := Organize Γ
6: τ_N := Normalize τ
7: root := Create inhabitation node τ_N
8: Enqueue root in Q
9: Spawn n working threads executing the task={
10:
11: while Q has elements and (not termination signal received) do
12:   Increase semaphore S
13:   q := Dequeue from Q (using assignment strategy S)
14:   if q is part of cycle then
15:     Mark q as end point of cycle
16:   end if
17:   τ̄' := Organize τ in q
18:   if τ̄' ∈ C_S then
19:     Link parent of τ̄' to existing τ̄' ∈ C_S
20:   else
21:     if ∃σ ∈ C_F.τ̄' ≤ σ then
22:       Mark τ̄' as FAILED
23:     end if
24:     if τ̄' ∉ C_F then
25:       if not InhabOptimized(Γ̄,τ̄') then
26:         Add τ̄' to C_F
27:         Mark τ̄' as FAILED
28:       end if
29:       Propagate result of τ̄' in execution graph G_E
30:     end if
31:   end if
32:   Decrease semaphore S
33: end while
34: }
35:
36: Wait for all n working threads
37: Fix cycles recursively in root
38: Mark UNKNOWN nodes as FAILED and propagate result
39: if root is marked with SUCCESS then
40:   return ACCEPT
41: else
42:   return FAIL
43: end if
```

Letia et al. [2009], has also been tested. An CRDT graph implementation turned out to be very efficient with respect to communication efforts needed for consesus on consistency because CRDTs do not induce conflicts during convergence. Therefore, CRDTs are conflict-free data structures. However, CRDT graphs also raised a problem of a slow convergence of eventually consistence that lead to unnecessary and redundant computations in an execution graph.

The concurrent algorithm is used for synthesis in practical scenarios. Some additional features, for example providing interfaces to the algorithm or an execution context, are needed to to simplify its usage. The resulting tool is presented in the next chapter.

Chapter 5

Combinatory Logic Synthesizer

The current chapter presents an implementation of the discussed concurrent inhabitation algorithm including the heuristic optimizations described in the previous chapter. The implementation is embedded in a tool named Combinatory Logic Synthesizer (CL)S and will be discussed now. We begin with the construction algorithm of resulting inhabitants of a relativized inhabitation request from an execution graph, then (CL)S implementation and relevant features are presented. Afterwards, (service-)interfaces of (CL)S for input and output are defined and some application scenarios, to which (CL)S and an extension of (CL)S, SCS, have been employed, are exhibited.

The distributed algorithm presented in Section 4.3 was implemented for $\mathsf{BCL}_0(\cap, \leq)$ using Microsoft .NET Framework. The data-centric part of the inhabitation algorithm (cf. Algorithm 4.7 on page 96) was implemented in F# whereas the control-centric part of the inhabitation algorithm (cf. Algorithm 4.8 on page 97) was implemented in C#. The core of the F# algorithm is shown in the appendix in Listing A.4.1 on page 205.

The non-deterministic choices in N_I have been implemented by iterations on all possible choices. For example, Algorithm 4.7 on page 96 iterates every $x \in \Gamma$ in line 3 and checked if its target is a subtype of τ in line 13. The algorithms discussed before are algorithms for deciding the relativized inhabitation problem. A subtle but important difference of the algorithms presented now is that (CL)S explicitly enumerates and returns all inhabitants.

5.1 Reconstructing Inhabitants

Inhabitants have to be constructed explicitly by decompressing the compressed information contained in the execution graph because a sharing representation to the execution graph is used. The reconstruction of inhabitants in an

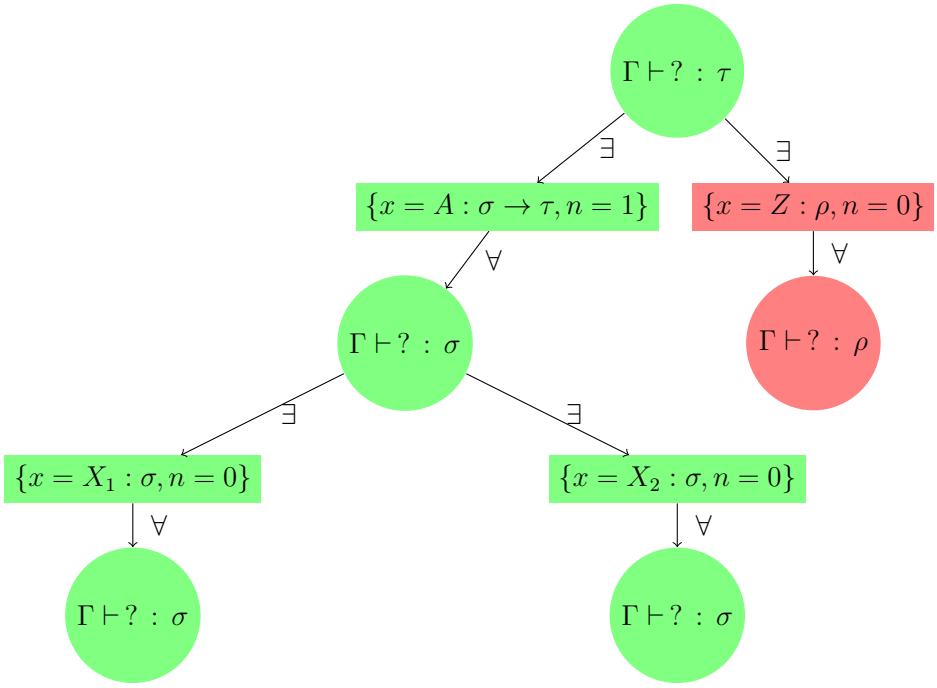


Figure 5.1: Constructing inhabitant from an execution graph $\mathcal{G}_E$

execution graph is achieved by following all paths from the root to successful leafs in the constructed execution graph and by storing every combinator x that occurs in a group node on this path. The following example will clarify this idea.

Example 5.1. *An example execution graph $\mathcal{G}_E$ is shown in Figure 5.1. Green nodes are marked with SUCCESS and red nodes are marked with FAILED. In this execution graph two leaves are marked with SUCCESS. Two paths to these nodes can be found from the root. The group nodes contain the combinators $x = A : \sigma \to \tau$, $x = X_1 : \sigma$, and $x = X_2 : \sigma$. The resulting inhabitants for the initial inhabitation question $\Gamma \vdash ? : \tau$ are AX_1 and AX_2.*

The algorithm for the construction is displayed as recursive Algorithm 5.1. The initial call to this algorithm is made with the root node of the execution graph $\mathcal{G}_E$ and returns the set of all inhabitants as applicative terms. This algorithm constructs sets of inhabitants, by way of example the applicative term $A(X_1(X_2), X_3(X_4, X_5))$ is such an inhabitant.

5.2 Implementation

(CL)S is a direct implementation of the Algorithms 4.7 and 4.9. (CL)S

Algorithm 5.1: Algorithm for reconstruction of inhabitants `Reconstruct`

Require: n is inhabitation node in $\mathcal{G}_E$

1: $S := \emptyset$
2: **for all** g is child group node of n **and** g is marked with *SUCCESS* **do**
3: $S' := \emptyset$
4: $x :=$ combinator of g
5: $i := 1$
6: **for all** n' is child inhabitation node of g **and** n' is marked with *SUCCESS* **do**
7: $arg_i :=$ `Reconstruct`(n') is x i-th argument
8: $S' := S' \cup \{x(arg_1, \ldots, arg_n)$
9: $i := i + 1$
10: **end for**
11: $S' := S' \cup S''$
12: **end for**
13: **return** S

supports two different processing modes that have both been compiled for Microsoft Windows as well as for Linux (using Mono). These are a batch-mode processing a synthesis request from a local file. A batch mode is important for automatically conducting experiments. A webservice-mode for synthesis requests on a remote server exposes endpoints offering access via SOAP and REST. To this end, there are two hosting solutions of the webservice. First, there is a stand-alone server, mainly intended for usage in experiments. Second, there exists a hosted version for application servers (e.g. Microsoft Internet Information Server IIS), intended for usage in industrial settings.

(CL)S as a server application provides its inhabitation service via a web service interface (WS-SOAP/REST/NetTCP). The inhabitation service implements a method Inhabitate with a string (containing the inhabitation request) as argument and returns a unique string containing a token that is used for polling the inhabitation results. The calculation of the inhabitation question takes undeterminable time and therefore an algorithm using a busy-waiting approach might produce a time-out error. Calling the method GetResults with the token as argument has two possible results. Either an empty string, meaning there is no inhabitation result yet, or the inhabitation result as a string containing an XML-document is returned.

The software architecture of (CL)S as a layer diagram is depicted in Figure 5.2 on the following page. The service provides auxiliary standard

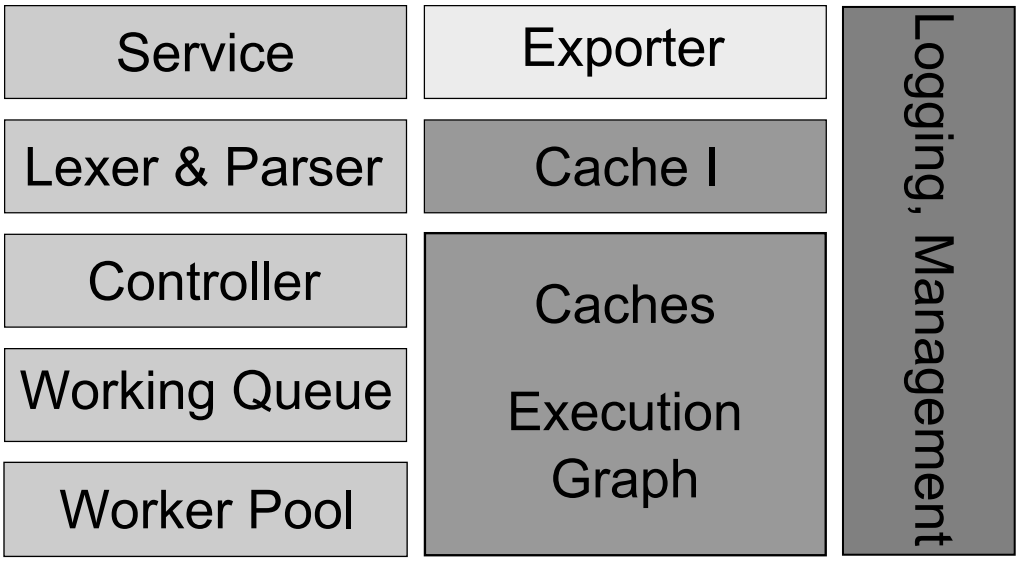


Figure 5.2: (CL)S architecture as layer diagram

features like thread and application pooling as well as security. Configurable logging functionalities are provided by (CL)S which is a crucial feature with regard to debugging and analyzing experiments.

The external communication with (CL)S is done through its service interfaces at the top in Figure 5.2, its exporting interfaces, or its logging files. The results are exported in various formats discussed in the (CL)S output section. For analysis, the execution graph and solution terms can be outputted graphically.

A component diagram of the implementation of (CL)S is depicted in Figure 5.3. In Figure 5.4 the inter-object dependencies and associations are shown. Architectural standard techniques like a loadbalancer and an additional application cache for filtering redundant calculations are omitted for clarity in front of the controller component in Figure 5.4. The execution graph is implemented as a shared memory data structure with spin locks. An alternative implementation as CRDT graph is possible. The working queue is also replaceable by a distributed queue, e.g. by using ∅MQ, in a distributed compute cluster. Moderate resulting communication costs, primarily resulting from interconnect latency of $\approx 20\mu s$, by using ∅MQ as distributed queue turned out to be a promising solution in a distributed compute cluster and replaces the work-stealing queue in a many- and multi-core implementation in the used compute cluster with 19 cluster units and more than 1200 cores.

The server allows multiple clients to share the server's threads in a fair

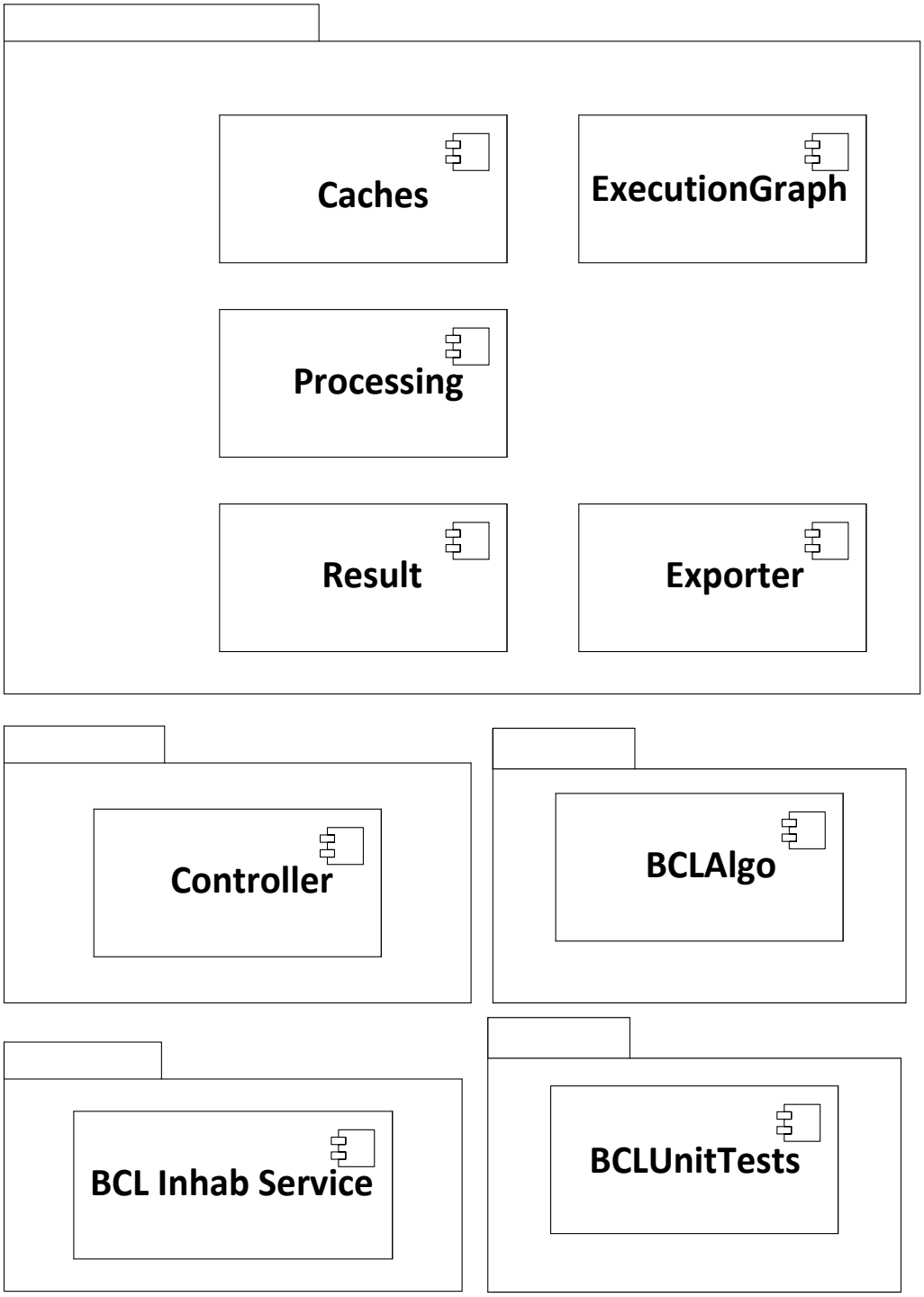


Figure 5.3: (CL)S package diagram

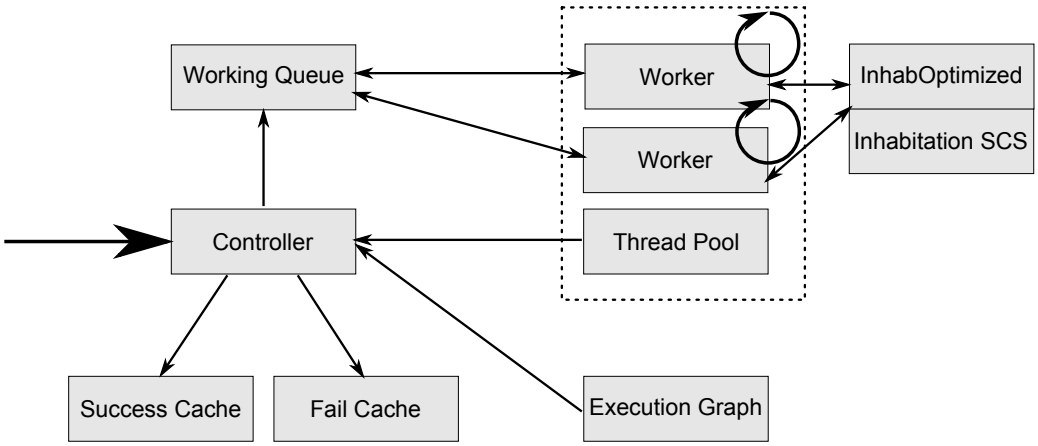


Figure 5.4: (CL)S inter-object dependencies

way. Because every client request is associated with a fixed number of threads that depends on the number of CPU cores of the system. Therefore, every core is shared equally.

5.3 (CL)S Input Specification

In the previous subsection, the input specification of (CL)S for an inhabitation request was introduced as a string. The string is a *domain-specific language* (DSL) aligned to the mathematical specification of $\mathsf{BCL}_k(\cap, \leq)$ presented in Section 3.3 on page 39. We prefer a DSL instead of an *embedded domain-specific language* (eDSL), because an eDSL would require an individual compilation for every novel input of (CL)S disallowing a batch processing mode of (CL)S.

Table 5.1 on the next page depicts mathematical operators and their corresponding (CL)S operator representation used in the (CL)S input grammar. The correspondence between both representations is intended to be close.

The data structure representation of intersections is a recursive list of intersection types. A list structure is an appropriate representation, because $\cap$ is associative (see Section 3.3 on page 39). This representation corresponds to the original representation of intersection types in [Barendregt et al., 1983].

	Mathematical example	(CL)S representation example
Atoms	τ, σ	`tau`, `sigma`, τ, σ
Variables	α, β	`alpha`, `beta`, α, β
$\rightarrow$	$\tau \rightarrow \sigma$	`tau->sigma` or $\tau \rightarrow \sigma$
$\cap$	$\tau \cap \sigma$	`[tau, sigma]` or `[` τ `,` σ`]`
Covariant constructor	$C(\tau_1, \ldots, \tau_n)$	`C(tau1,...,taun)`
$\leq$	$\tau \leq \sigma$	`tau<=sigma` or $\tau \leq \sigma$
Subst.	$S(\alpha) = \tau$	$\{\alpha\} => \{\tau\}$ $\{\alpha\} \sim> \{\tau\}$

Table 5.1: Mathematical operators and corresponding expressions in (CL)S

Example 5.2. *Assuming a type environment*

$$\begin{aligned}\Gamma = \quad \{ & \\ & A : \sigma_1 \rightarrow \sigma_2 \cap \sigma_3 \rightarrow a, \\ & B : \alpha, \\ & C : \sigma_2 \cap \sigma_3 \\ \}, &\end{aligned}$$

with the substitution $\{\alpha \mapsto \sigma_1\}$ *and the atomic subtyping extension* $a \leq a'$ *with type atoms* a *and* a'*. Then* $\Gamma \vdash ? : a'$ *can be coded as input for* (CL)S *as follows:*

```
{
   (* Type environment Gamma *)
   A : sigma1 -> [sigma2, sigma3] -> a,
   B : alpha,
   C : [sigma2, sigma3]
},
{
   (* Substitution(s) *)
   {alpha} => {sigma1}
},
{
   (* Atomic subtyping extension *)
   tau<=tau'
}
|- ? : a' (* Inhabitation question *)
```

Appendix A.1 on page 199 contains the grammar definition of (CL)S in *extended Backus-Naur form* (EBNF).

This input is parsed into an *abstract syntax tree* (AST) that is similar to the functional data-type representation of (CL)S. This data-type representation is used by the implementation of the inhabitation algorithm in Figure 4.9 on page 102 and a F# implementation in Listing A.1 on page 205.

5.4 (CL)S Output Specification

(CL)S returns inhabitants of the posed inhabitation request as a string containing an XML document with all inhabitants as XML tree and auxiliary information. Additionally, the execution graph and the contained applicative terms are exported to a *directed graph markup language* (DGML)[1] file for graphical representation. DGML is a graph description language using XML containing a directed graph with additional information. This information can be used to filter or operate on the graph in Microsoft Visual Studio 2013. These operations include queries on the graph structure and content or layout operations like cluster view or tree view.

In different experiments conducted with (CL)S, graphs have been created with more than 1 000 000 nodes. For analysis and debugging purposes this graph representation of the execution graph $\mathcal{G}_E$ has proven very valuable.

The DGML representation is graphically similar to the execution graph $\mathcal{G}_E$ in Section 4.2.1. Figure 5.5 and Figure 5.6 show an example of a small execution graph and a more complex execution graph as output in DGML. The example in Figure 5.5 is taken from the tracking object-example presented in [Rehof, 2013]. In Figure 5.6, the extended tracking object-example presented in [Düdder et al., 2014a, 2013b] and in Section 1.3 on page 7 is depicted.

Another export filter of (CL)S for execution graphs is the DOT format. DOT is also a graph representation language. GraphViz[2] uses the DOT format as input to render graphic files in various formats. Open-source editors and viewers for DOT are available. This file format is particularly important with regard to Linux users.

Furthermore, various data of the processing of (CL)S are logged in files. This information includes timings, CPU and thread utilization, RAM consumption, and warnings and errors.

[1] XML-schema definition of DGML: `http://schemas.microsoft.com/vs/2009/dgml`

[2] GraphViz's and Dot's main page is `http://www.graphviz.org`.

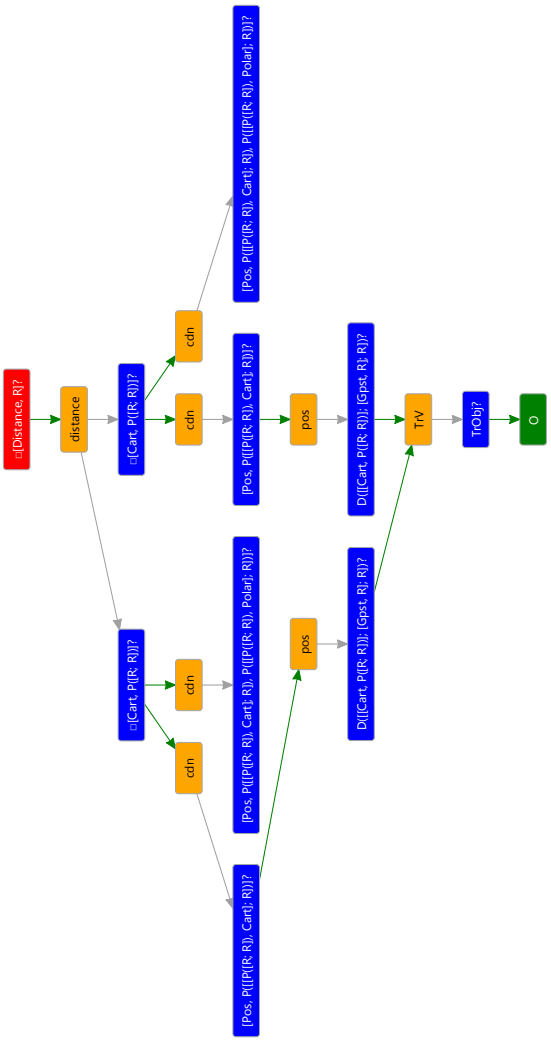


Figure 5.5: (CL)S execution graph in DGML

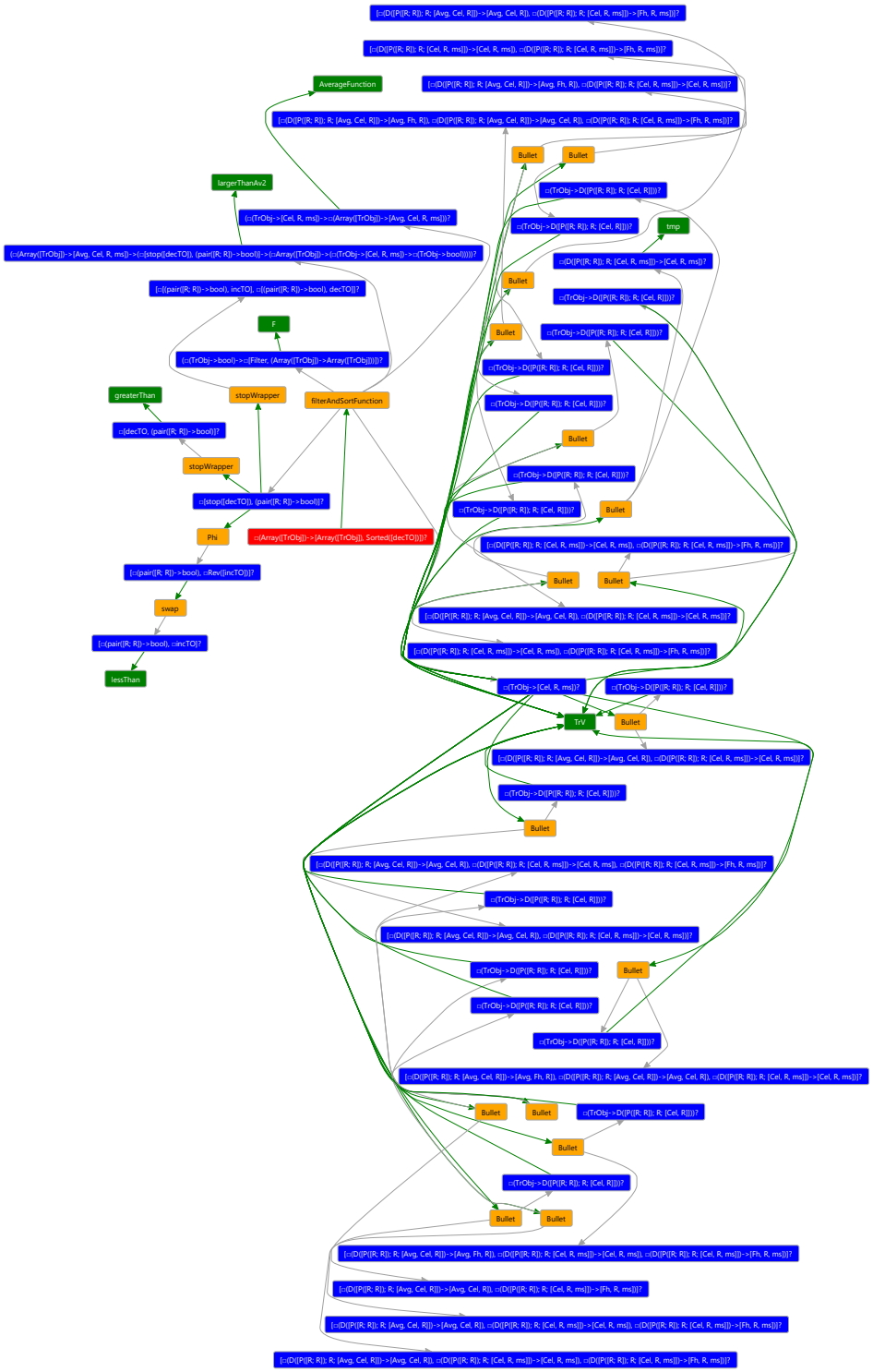


Figure 5.6: (CL)S execution graph in DGML

5.5 Additional Applications and Extensions

(CL)S has been employed to various synthesis scenarios:

1. In Combinatory Process Synthesis (CPS), Vasileva [2013] uses SCS[3] as a tool to generate BPMN 2.0 workflows (Activiti[4]) from a repository of process components.

2. Wolf [2013] uses (CL)S to synthesize control programs for LEGO® Mindstorm NXT robots from a repository of atomic and complex control components .

3. Plate [2013] uses (CL)S to synthesize configurations for virtual machine images in cloud computing infrastructures with OpenNebula[5] as cloud computing stack and corresponding deployment code for instantiating synthesized virtual machine images from a repository containing various configuration components and a comprehensive IT infrastructure taxonomy.

4. Düdder et al. [2014b] use (CL)S featuring SCS for automatically synthesizing deployable and executeable business process workflows in BPMN 2.0 for a specific workflow management system (Activiti).

5. (CL)S with SCS is used by Bessai et al. [2014a] for synthesizing compositions of mixins in object-oriented software systems for Java 8.

6. Bessai et al. [2014b] provides an overview of features of (CL)S that has been developed in this thesis and features an example of synthesizing Dependency Injection code for Java and Spring framework[6].

In order to simplify the design and editing of type environments, (CL)S ships with a configurable editor extension(cf. Figure 5.8) for Microsoft Visual Studio 2013 providing syntax highlighting, code completion, support for revision control and source code management (CVS, subversion and GIT) as well as sending inhabitation requests directly at the push of a button to a (CL)S service. Furthermore, an extension for the open-source programming editor `Notepad++`[7], depicted in Figure 5.7, provides syntax highlighting and also code completion for various operating systems, for example Microsoft

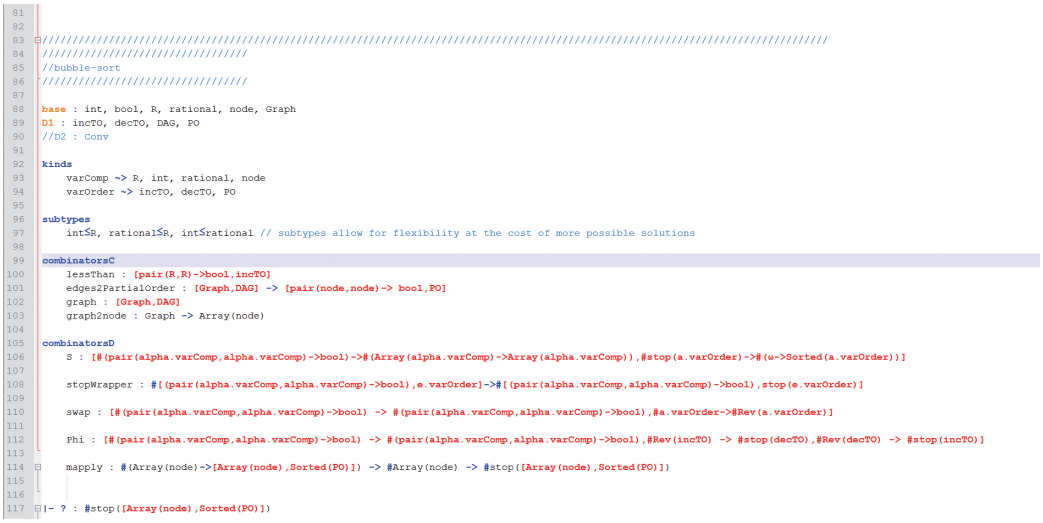
[3] SCS is an extension of (CL)S for staged computation synthesis Düdder et al. [2014a] (see for 7.6 on page 159)

[4] `http://activiti.org`

[5] `http://opennebula.org/`

[6] `http://spring.io/`

[7] `http://notepad-plus-plus.org/`



```
81
82
83  ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
84  ////////////////////////////////////
85  //bubble-sort
86  ////////////////////////////////////
87
88  base : int, bool, R, rational, node, Graph
89  D1 : incTO, decTO, DAG, PO
90  //D2 : Conv
91
92  kinds
93      varComp ~> R, int, rational, node
94      varOrder ~> incTO, decTO, PO
95
96  subtypes
97      int≤R, rational≤R, int≤rational // subtypes allow for flexibility at the cost of more possible solutions
98
99  combinatorsC
100     lessThan : [pair(R,R)->bool,incTO]
101     edges2PartialOrder : [Graph,DAG] -> [pair(node,node)-> bool,PO]
102     graph : [Graph,DAG]
103     graph2node : Graph -> Array(node)
104
105 combinatorsD
106     S : [#(pair(alpha.varComp,alpha.varComp)->bool)->#(Array(alpha.varComp)->Array(alpha.varComp)),#stop(a.varOrder)->#(ω->Sorted(a.varOrder))]
107
108     stopWrapper : #[(pair(alpha.varComp,alpha.varComp)->bool),e.varOrder]->#[(pair(alpha.varComp,alpha.varComp)->bool),stop(e.varOrder)]
109
110     swap : [#(pair(alpha.varComp,alpha.varComp)->bool) -> #(pair(alpha.varComp,alpha.varComp)->bool),#a.varOrder->#Rev(a.varOrder)]
111
112     Phi : [#(pair(alpha.varComp,alpha.varComp)->bool) -> #(pair(alpha.varComp,alpha.varComp)->bool),#Rev(incTO) -> #stop(decTO),#Rev(decTO) -> #stop(incTO)]
113
114     mapply : #(Array(node)->[Array(node),Sorted(PO)]) -> #Array(node) -> #stop([Array(node),Sorted(PO)])
115
116
117 |- ? : #stop([Array(node),Sorted(PO)])
```

Figure 5.7: (CL)S support in the open source programming editor `Notepad++`

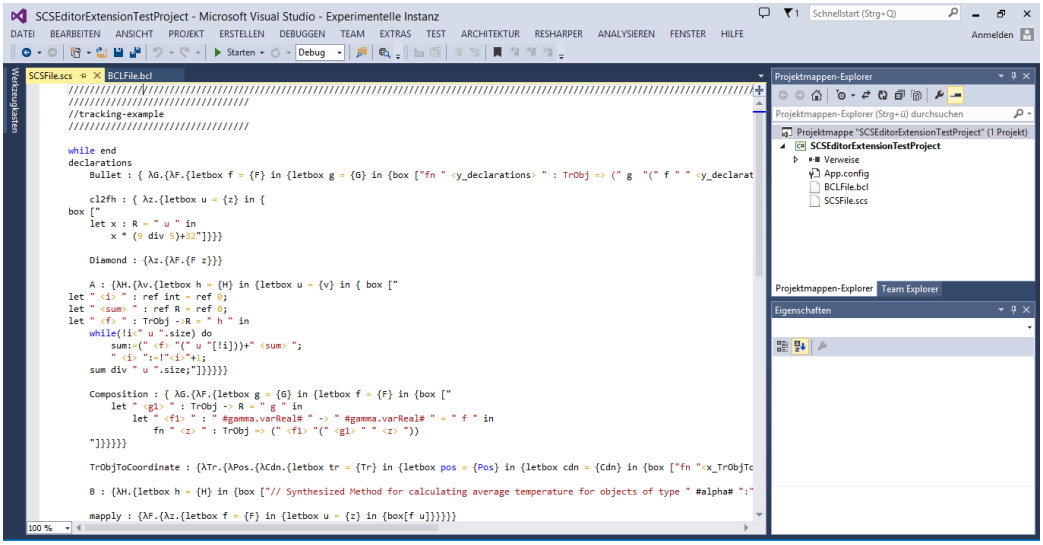


Figure 5.8: SCS support in Microsoft Visual Studio 2013

```
largerThanAv2 : {λAverageFunction.
                    {λOrder.
                        {λArray.
                            {λTr2TmpFunction.
                                {letbox av = {λAvFn.{λAr.{letbox avFn = {AvFn} in
                                    {letbox ar = {Ar} in {box{avFn"("ar")"}}}}} AverageFunction Array} in
                                        {letbox order = {Order} in
                                            {letbox tr2TempFn = {Tr2TmpFunction} in
                                                {box{"fn "<x_largerThanAv2>" : "#alpha.varTr#" => not("order"("av","tr2TempFn"("<x_largerThanAv2>")))"}}}}}}}}}

filterAndSortFunction : {λConvertTrObjToTemp.
                            {λOrder.
                                {λCalculateAverageTempGenerator.
                                    {λPredConstructor.
                                        {λFilterGenerator.
                                            {letbox trObj2Temp = {ConvertTrObjToTemp} in
                                                {letbox calculateAverageTemp = {CalculateAverageTempGenerator box ["tr2TempFunction_filterAndSortFunction"]} in
                                                    {letbox pred = {PredConstructor box["calculateAverageTempFunction_filterAndSortFunction"] box[
                                                    "orderFunction"] box[<trObjArray_filterAndSortFunction>] box["tr2TempFunction_filterAndSortFunction"]} in
                                                        {letbox filter = {FilterGenerator box["averageFilterPredicate"]} in
                                                            {letbox order = {Order} in
                                                                {box {"
fn "<trObjArray_filterAndSortFunction>" : Array(" #alpha.varTr# ") =>
    let tr2TempFunction_filterAndSortFunction : "#alpha.varTr#" -> "#alpha.varComp#" = "trObj2Temp"
    in
    let orderFunction : (" #alpha.varComp# "," #alpha.varComp# ") -> bool = "order"
    in
    let calculateAverageTempFunction_filterAndSortFunction : Array(" #alpha.varTr# ") -> "#alpha.varComp#" =
        "calculateAverageTemp"
    in
    let averageFilterPredicate : "#alpha.varTr#" -> bool = "pred"
    in
    let filterByAverage_filterAndSortFunction : Array(" #alpha.varTr# ") -> Array(" #alpha.varTr# ") = "filter"
    in
    let swap : (Array(" #alpha.varTr# "),int,int) -> Array(" #alpha.varTr# ") =
        fn ("<xSwap>",leftIndex,rightIndex) : (Array(" #alpha.varTr# "),int,int) =>
            let temp : ref " #alpha.varTr# " := "<xSwap>"[rightIndex];
            "<xSwap>"[rightIndex] := "<xSwap>"[leftIndex];
            "<xSwap>"[leftIndex] := !temp;
    in
```

Figure 5.9: SCS support in the open source programming editor `Notepad++`

Windows and Linux. Both editor extensions also support the (CL)S extension SCS as depicted in Figure 5.9.

Chapter 6

Synthesis of Software Architectures

This chapter contains a detailed discussion on software architecture and a presentation of the Combinatory Logic Connector Synthesis method. We develop a set-theoretic model for capturing relevant features and relations of software architectural entities. Using this set-theoretic model and composition in mind, we will identify and classify various common building blocks as well as their type-theoretic encoding for later use in a repository defined as a type environment. We methodologically develop the idea of taxonomic hierarchies encoding semantic concepts of software architecture and present an excerpt of a comprehensive taxonomy used in various experiments.

Constituting on this work, we define the Combinatory Logic Connector Synthesis method as specialization of combinatory logic synthesis and devise its consisting sequential steps. These steps consist of the following major activities: definition of a connector synthesis goal, encoding existing connectors as intersection types and collecting these intersection types in a designated connector type environment, application of relativized type inhabitation for a connector type environment and a connector synthesis goal, and generation of objects, e.g. compilable and executable program code, from synthesis results. Afterwards, several development and design best practices for connector type environments are presented.

6.1 Software Connectors

We recall the notions of components and connectors in software architecture [Perry and Wolf, 1992, Allen and Garlan, 1997, Taylor et al., 2010] and discuss our methodology and the underlying type-theoretic model. The figures will

use UML2 component diagrams as ADL by default. The connectors developed in this chapter allow for a syntactic *and* semantic interconnection.

6.2 Type-theoretic Model

We explain how intersection types are used to specify components, connectors, and building blocks. Let P be a finite set of semantic concepts, ranged over by p, p', that describe properties of components, connectors, building blocks, and interfaces. Let $\mathcal{T} \subseteq P \times P$ be a preorder on P forming a taxonomical hierarchy. Let $\mathcal{I}$ be a finite set of interfaces ranged over by I, I'. Denote by $\mathbb{T}_{P,\mathcal{I}}$ be the set of intersection types that can be built from taking $P \cup \mathcal{I}$ to be the set of constants. We denote $\tau \in \mathbb{T}_{P,\mathcal{I}}$ as a *connector type*. We extend the subtyping relation $\leq$ on $\mathbb{T}_{P,\mathcal{I}}$ by $\mathcal{T}$ by adding the axiom $(p, p') \in \mathcal{T} \Rightarrow p \leq p'$ to the axioms given in [Barendregt et al., 1983]. We will also say, that a set of axioms is added to the subtyping relation $\leq$. The extension of the subtyping relation is used to encode semantic concepts in a taxonomical hierarchy that are specifying the building blocks of software connectors.

Remark 6.1. *Note that interfaces of components and connectors are more abstract in the sense that these interfaces can be seen as describing a set of methods with a unique symbol instead of a single method with a method name.*

We begin with a set theoretic-model of software components and software connectors that eases the discussion and the construction of a type-theoretic model that our method is based on.

6.2.1 Component

In our set-theoretic model, a software component is characterized by its interaction via its provided and required interfaces. The component's computation is abstract and hidden because we are only interested in synthesizing software connectors.

Definition 6.2.1. ***(Component)***
A component C *is a pair* $\mathsf{C} = (\mathcal{R}, \mathcal{P})$ *where* $\mathcal{R}, \mathcal{P} \subseteq \mathcal{I}$ *with* $\mathcal{P} \neq \emptyset$ *are* C*'s required resp. provided interfaces.*

Each $I_j \in \mathcal{R}$, $\mathcal{P}$, and C itself may be annotated by sets of concepts P_j, $P_{\mathcal{P}}$, $P_{\mathsf{C}} \subseteq P$. Some of the sets of concepts may be empty. This does not present a problem in the following since the empty intersection is equal to ω and it can be shown that $\tau \cap \omega = \tau$ holds. A software component can be translated into an intersection type representing this component as follows.

Definition 6.2.2. *(**Component type**)*
Let C *be a component with* $\mathcal{R} = \{I_1, \ldots, I_k\}$. *Then,* C *is represented by the type* $\tau_{\mathsf{C}} \equiv (\tau_1 \to \ldots \to \tau_k \to \tau_{\mathcal{P}}) \cap \bigcap_{p \in P_{\mathsf{C}}} p$ *where* $\tau_j \equiv I_j \cap \bigcap_{p \in P_j} p$ *for* $1 \leq j \leq k$ *and* $\tau_{\mathcal{P}} \equiv \bigcap_{I \in \mathcal{P}} I \cap \bigcap_{p \in P_{\mathcal{P}}} p$.

6.2.2 Connector

In our set-theoretic model, a software connector is characterized by its interaction via its provided and required interfaces as well as the functional dependency between provided and required interfaces. A software connector might need some external services that are connected to its required interfaces in order to provide some of its services by its provided interfaces. These external services might be provided by supplemental software components.

Definition 6.2.3. *(**Connector**)*
A connector c *is a triple* $\mathsf{c} = (\mathcal{R}, \mathcal{P}, f)$ *where* $\mathcal{R}, \mathcal{P} \subseteq \mathcal{I}$ *with* $\mathcal{R} \neq \emptyset$ *and* $\mathcal{P} \neq \emptyset$ *describe* c*'s required resp. provided interfaces. The partial function* $f : 2^{\mathcal{P}} \to 2^{\mathcal{R}}$ *maps subsets of provided interfaces of* c *to subsets of required interfaces of* c.

We intuitively explain the meaning of f. Let $\mathcal{P}' \subseteq \mathcal{P}$ be a set of provided interfaces of c. Then, $f(\mathcal{P}')$, if defined, states which of the required interfaces of c are needed to provide the functionality of the interfaces in $\mathcal{P}'$. A connector c may connect a set $\mathcal{P}_1 \subseteq \mathcal{I}$ of provided interfaces to a set $\mathcal{R}_1 \subseteq \mathcal{I}$ of required interfaces if all interfaces that c requires to provide $\mathcal{R}_1$ are contained in $\mathcal{P}_1$. Thus, c must be able to provide the interfaces in $\mathcal{R}_1$ (i.e., $f(\mathcal{R}_1)$ must be defined). Furthermore, the interfaces that c requires to do so must be provided to c (i.e., $f(\mathcal{R}_1) \subseteq \mathcal{P}_1$).

Definition 6.2.4. *(**May connect**)*
c may connect $\mathcal{P}_1$ to $\mathcal{R}_1$ *if the following two conditions hold:*

1. $f(\mathcal{R}_1) \neq \bot$ *and*
2. $f(\mathcal{R}_1) \subseteq \mathcal{P}_1$.

Note that an I-connector is represented by $(\{I\}, \{I\}, \{\{I\} \mapsto \{I\}\})$. To translate a connector c to a typed combinator, consider a set $\mathcal{P}' \subseteq \mathcal{P}$ of provided interfaces with $f(\mathcal{P}') = \{I_1, \ldots, I_k\}$. Again, each I_j where $1 \leq j \leq k$ and $\mathcal{P}'$ may be annotated by sets P_j resp. $P_{\mathcal{P}'} \subseteq P$ of concepts. We represent this functional dependency by the type $\sigma_{\mathcal{P}'} \equiv \tau_1 \to \ldots \to \tau_k \to \tau_{\mathcal{P}'}$ where $\tau_j \equiv I_j \cap \bigcap_{p \in P_j} p$ for $1 \leq j \leq k$ and $\tau_{\mathcal{P}'} \equiv \bigcap_{I \in \mathcal{P}'} I \cap \bigcap_{p \in P_{\mathcal{P}'}} p$. The connector c may also be annotated by a set $P_{\mathsf{c}} \subseteq P$ of concepts.

A software connector can be translated into an intersection type representing this connector as follows.

Definition 6.2.5. *(**Connector type**)*
The connector is represented by the type $\tau_{\mathtt{c}} \equiv \bigcap_{\mathcal{P}' \subseteq \mathcal{P}} \sigma_{\mathcal{P}'} \cap \bigcap_{p \in P_{\mathtt{c}}} p$.

In this definition, we tacitly assume that the first intersection is only indexed by subsets $\mathcal{P}'$ with $f(\mathcal{P}') \neq \bot$.

6.2.3 Building Blocks

We now define some identified building blocks and provide examples for their definition. These building blocks classify characteristic usages of software connectors. This classification system is an integral part of our method and contains the following elements

- *Atomic* building block
- *Complex* building block
- *Container* building block
- *Adapter* building block

Atomic Building Block

A very basic building block is a building block that only provides interfaces without requiring any interfaces. Such an atomic building block does not need any other building blocks to provide its service via its provided interfaces.

Definition 6.2.6. *(**Atomic building block**)*
Let $\mathtt{b}$ *be an* atomic building block *providing the interfaces in* $\mathcal{P} \subseteq \mathcal{I}$ *which may be annotated by a set* $P_{\mathtt{b}} \subseteq P$ *of concepts. It is represented by the type* $\tau_{\mathtt{b}} \equiv \bigcap_{I \in \mathcal{P}} I \cap \bigcap_{p \in P_{\mathtt{b}}} p$.

Example 6.1. *Assume an atomic building block* $\mathtt{b}$ *as depicted in Figure 6.1 providing three interfaces with* $\mathcal{P}_{\mathtt{b}} = \{I_1, I_2, I_3\}$. *Furthermore, assume that we have concepts* $P_{\mathtt{b}} = \{p_1, p_2\}$ *associated to* $\mathtt{b}$. *The representation of* $\mathtt{b}$ *as type is:*

$$\tau_{\mathtt{b}} \equiv I_1 \cap I_2 \cap I_3 \cap p_1 \cap p_2.$$

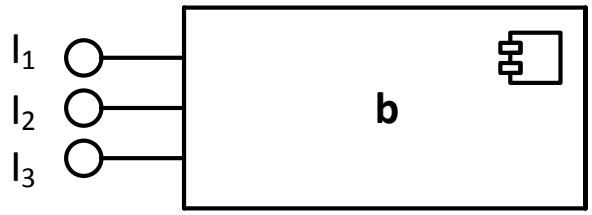


Figure 6.1: Atomic building block

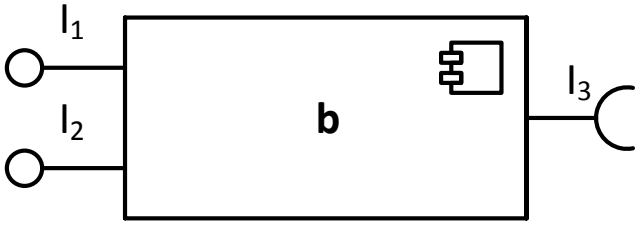


Figure 6.2: Complex building block

Complex Building Block

Definition 6.2.7. *(**Complex building block**)*
A complex building block $\mathtt{b}$ *may be described by an arbitrary type* $\tau_{\mathtt{b}} \in \mathbb{T}_{P,\mathcal{I}}$.

Example 6.2. *Assume a complex building block* $\mathtt{b}$ *depicted in Figure 6.2 providing two interfaces with* $\mathcal{P}_{\mathtt{b}} = \{I_1, I_2\}$ *and requiring one interface with* $\mathcal{R}_{\mathtt{b}} = \{I_3\}$. *Furthermore, assume that we have concepts* $P_{\mathtt{b}} = \{p_1, p_2\}$ *associated to* $\mathtt{b}$. *The representation of* $\mathtt{b}$ *as type is:*

$$\tau_{\mathtt{b}} \equiv (I_1 \cap I_2) \to I_3 \cap p_1 \cap p_2.$$

Thus, complex building blocks are the only objects whose types may contain type variables. Such variables are used to connect a complex building block to concrete interfaces. This definition allows us to regard components and connectors as building blocks. A special kind of complex building blocks is used as top-level containers that offers the functionality of a software connector and packages all its required service components.

Container Building Block

Definition 6.2.8. *(**Container building block**)*
A building block $\mathtt{b}$ *whose type* $\tau_{\mathtt{b}}$ *has at least one type component of the form* $\sigma_1 \to \ldots \to \sigma_m \to (\iota_1 \cap \rho_1 \to \ldots \to \iota_l \cap \rho_l \to \bigcap_{k \in K} \iota'_k \cap \rho)$ *where* $m \geq 0$ *and* $l, |K| > 0$ *is called a* container (building block).

Here, ι_i resp. ι'_j are atomic type variables that are kinded by $\mathcal{I}$ (see for Section 3.2.3 on page 39). Type variables ι act as meta-variables that get instantiated with concrete interfaces of components. The types ρ_h for $1 \leq h \leq l$ and ρ are level-0 types that can be built from P and type variables kinded by P. Such containers can be regarded as the top-level building blocks from which connectors can be synthesized. The types σ_i can be regarded as the arguments required to synthesize the connector.

Example 6.3. *If, for example, there is a container* $(\mathtt{b} : I' \cap p \to (\iota \cap p' \to \iota \cap p''))$ *an* I*-connector can be synthesized (by instantiating* ι *by* I*) if* $\mathtt{b}$ *is provided with an interface* I' *that has been specified by concept* p*. This* I*-connector would have the property that it requires* I *to have property* p' *and then provides* I *with property* p''*.*

The container building block is an important building block for realistic models and also for synthesis. Container building blocks will later be represented as UML2-packaging components [Object Management Group (OMG), 2005] because container building blocks are logical (and often meta combinators in the sense of $L2$ in Section 7.6 on page 159) entities that resemble a specific, actual usage-context, e.g. technical environment (triggering tools), for child building blocks of such container building blocks.

Another example for a container building block is shown in the following example.

Example 6.4. *Assume a container building block* $\mathtt{b}$*, depicted in Figure 6.3, providing three interfaces with* $\mathcal{P}_\mathtt{b} = \{\iota, I_1, I_2\}$ *and requiring one interface with* $\mathcal{R}_\mathtt{b} = \{\iota\}$*. The interface* ι *is the placeholder (as type variable) for connecting this container building block to components. Furthermore, assume that we have two concepts* $P_\mathtt{b} = \{p_1, p_2\}$ *associated to* $\mathtt{b}$*. The representation of* $\mathtt{b}$ *as type is:*

$$\tau_\mathtt{b} \equiv I_1 \to I_2 \to ((\iota \to \iota) \cap p_1 \cap p_2).$$

Adapter Building Block

Definition 6.2.9. ***(Adapter building block)***
A building block $\mathtt{b}$ *whose type* $\tau_\mathtt{b}$ *of the form* $(\iota_1 \cap \rho_1 \to \ldots \to \iota_l \cap \rho_l \to \bigcap_{k \in K} \iota'_k \cap \rho)$ *where* $m \geq 0$ *and* $l, |K| > 0$ *is called an* adapter (building block)*.*

Here, ι_i resp. ι'_j are atomic type variables that are kinded by $\mathcal{I}$ (see for Section 4.1.5 on page 66). We will assume that an adapter building block is not an identity function and therefore P is not empty. The types ρ_h for

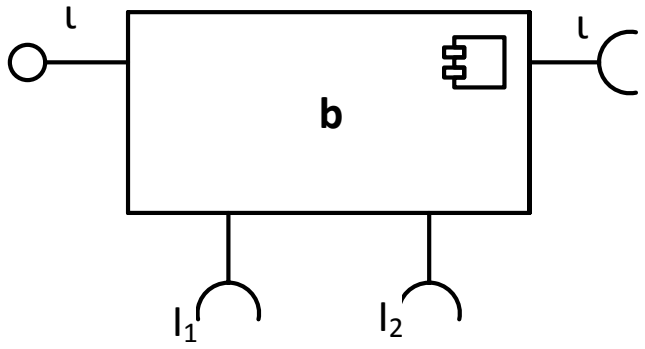


Figure 6.3: Container building block

$1 \leq h \leq l$ and ρ are level-0 types that can be built from P and type variables kinded by P. Such containers can be regarded as adapters for I-connector's with $\iota \to \iota$. In a later example and experiment in Section 8.2 on page 170, an adapter building block will be presenting a façade (pattern) [Buschmann et al., 1996, pages 294ff] to connect two slightly different interfaces. The adapter building block is a specialization of the container building blocks. These adapter building blocks can be derived from complex connectors by splicing the complex connector as a transformation as suggested in [Spitznagel and Garlan, 2001]. This transformation is an element of a proposed set of transformations that are discussed in Section 6.3.6.

One purpose of adapter building blocks is data transformation. Even though the components to be connected share the same interface ι, these interfaces might be specified by semantic information $\rho \in P$. An adapter building block can connect to other adapter building blocks and container building blocks.

Example 6.5. *Assume an adapter building block* $\mathtt{b}$ *providing one interface with* $\mathcal{P}_\mathtt{b} = \{\iota\}$ *and requiring one interface with* $\mathcal{R}_\mathtt{b} = \{\iota\}$*. Like in a container building block, the interface* ι *is the placeholder (as type variable) for connecting this adapter building block to components or other connectors. Furthermore, assume that we have concepts* $P_\mathtt{b} = \{p_1, p_2\}$ *associated to* $\mathtt{b}$*. The representation of* $\mathtt{b}$ *as type is:*

$$\tau_\mathtt{b} \equiv \iota \to \iota \cap p_1 \cap p_2.$$

6.2.4 C&C Type Environment

Let R be a given repository of building blocks.

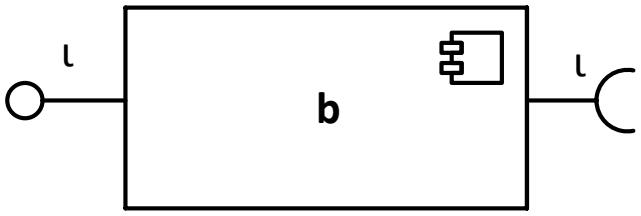


Figure 6.4: Adapter building block

Definition 6.2.10. ***(C&C type environment)***
A type repository containing at least one container building block is called C&C type environment*.*

We assume that R is a C&C type environment. For each C, c, and b in R we define typed combinators $(\mathsf{C} : \tau_{\mathsf{C}})$, $(\mathsf{c} : \tau_{\mathsf{c}})$, resp. $(\mathsf{b} : \tau_{\mathsf{b}})$ as above. We define the type environment Γ_R to contain exactly these typed combinators. $\mathsf{BCL}_0(\cap, \leq)$-inhabitation can be used to synthesize from R a connector c specified with certain concepts by asking the inhabitation question $\Gamma_R \vdash ? : \tau_{\mathsf{c}}$. A resulting inhabitant is an applicative term consisting of combinators representing building blocks in R. We define the inhabitation question and the interpretation of the applicative term.

6.2.5 Taxonomy

Taxonomies (or taxonomical hierarchies) are used to assign semantic concepts to objects by linking concepts coming from the taxonomy to those objects. These semantics can contain a rich structure as depicted in the Figure 6.5. The idea of encoding taxonomic trees into a subtyping relation is shown in [Rehof, 2013, pages 8f] and in [Steffen et al., 1997].

Our motivation to use taxonomies for the Combinatory Logic Connector Synthesis method is that we want to represent and automatically reason based on the software architect's knowledge on software architectures. Here, we will primarily focus on the conceptual knowledge on software connectors and the interaction among components.

The structure is given in the form of a taxonomic tree in Figure 6.5, in which the nodes contain semantic type names, and where dotted lines indicate structure containment. It means that Concept 2 is also a Concept 1. The Composite Concept is a composition of two concepts Concept 3 and Concept 4. Furthermore, we have an instance-of relation that relates abstract concepts with concrete instances. A taxonomy for connectors is presented and discussed in [Taylor et al., 2010]. If we directly transfer such a taxonomy

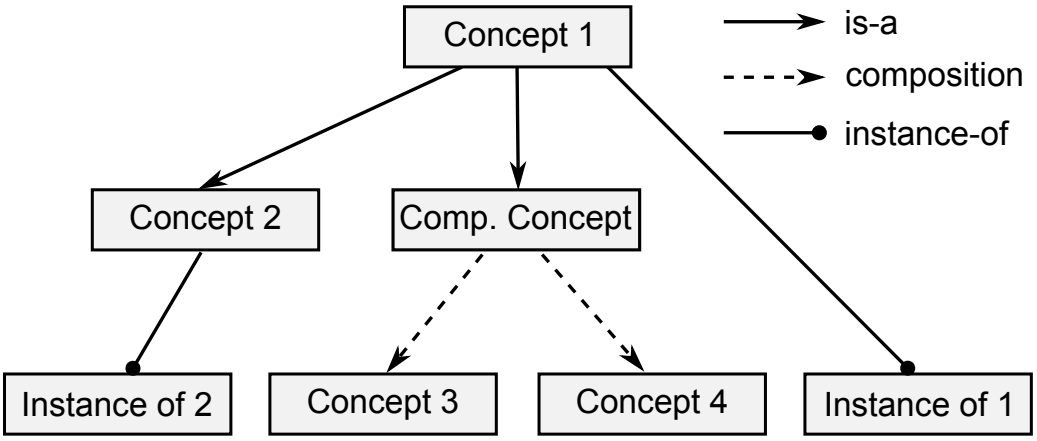


Figure 6.5: Example (abstract) taxonomy

into an atomic extension of the subtyping relation $\leq$, then we would lose the extra information of the semantic of the vertices. Within this relation we cannot distinguish between semantics of relations (is-a vs. instance-of) in a taxonomy. To integrate different semantics encoded by the corresponding relations, a separate meta-taxonomy for describing those semantics can be added to our subtyping relation $\leq$. A meta-taxonomy is depicted in Figure 6.6. This taxonomy only contains conceptual information on the semantics of taxonomic relations. To integrate both taxonomies into a subtyping relation,

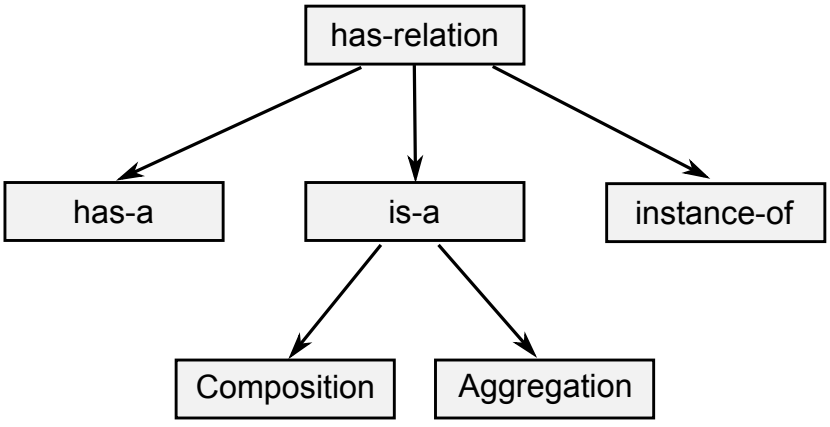


Figure 6.6: Example meta-taxonomy

1. we construct the atomic subtyping extension on the union of both relations.

2. we combine every occurrence of a taxonomic concept in a type of a combinator in the C&C type environment with the concept of its taxonomic relation originating from the meta-taxonomy. In the case of the composition relation, we use a coding using arrow types ($\rightarrow$). Assume the concept a is a composition of the concepts b, c, and d, then the relation is encoded as the type $b \rightarrow c \rightarrow d \rightarrow a$. This type is then inserted into the corresponding position in the combinator's type.

3. we encapsulate different taxonomical concepts in the arguments of designated type constructors.

This construction allows a succinct encoding of the taxonomical hierarchies into the subtyping relation $\leq$ and the C&C type environment. The atomic extension can easily be encoded in (CL)S as described in the previous chapter.

We can now use the connector classes and instances presented by Hirsch et al. [1999] depicted in Figure 6.1 to construct a taxonomy representing this information. The meaning of a connector class in [Hirsch et al., 1999] is: "[...] a connector class can be seen as a connector that does not define criteria for a set of properties, these properties are viewed as optional by the class. Consequently, a class instance is a connector with no optional properties that has all mandatory properties of its class." Asterisks are indicating optional properties. Pipe and Typed Pipe are instances of the class Pipe. The vertical line in Figure 6.1 on the facing page is used to visually separate instance classes like Typed Pipe from the classes like Pipe (class). PC and RPC abbreviate procedure call respectively remote procedure call. An idea for transferring these connector classes and instances to a taxonomy is to interpret properties such as to be "Connection oriented" as concepts and the instances either as concept instances (1) or directly as combinators (2) (including an underlying set of templates). In [Hirsch et al., 1999] an entry "Yes" means that this property is present whereas an entry "No" means that a property is not applicable. An entry "*" means that this property is optional. Here, we ignore the "No" and "*" cases and use "Yes" marked properties only. It means that the connector classes and instances in Figure 6.1 can be transferred into a taxonomy such as presented in Figure 6.7. The figure shows a taxonomical tree for the concept of a reliable connection, for instance a reliable connection can be implemented by a pipe.

We can now define the atomic subtyping extension of $\leq$ by adding the following axioms:

Property	Pipe	RPC	Event Broadcast	PC	Shared Data	Pipe	Typed Pipe
Knows target	No	Yes	No	Yes	No	No	No
Request/Reply	No	Yes	No	Yes	No	No	No
Synchronous	No	Yes	No	Yes	No	No	No
Flow Control	Yes	No	No	No	No	Yes	Yes
One Way	Yes	No	No	No	No	Yes	Yes
Broadcast	No	No	Yes	No	Yes	No	No
Streamed	Yes	No	No	No	No	Yes	Yes
Connection oriented	No	No	No	No	No	No	No
Reliable	Yes	Yes	Yes	Yes	No	Yes	Yes
Typed	*	Yes	Yes	Yes	Yes	No	Yes
Encryption	*	*	*	*	*	No	No
Authentication	*	*	*	*	*	No	No
Compression	*	*	*	*	*	No	No
Monitoring	*	*	*	*	*	No	No

Table 6.1: Connector classes and instances according to Hirsch et al. [1999]

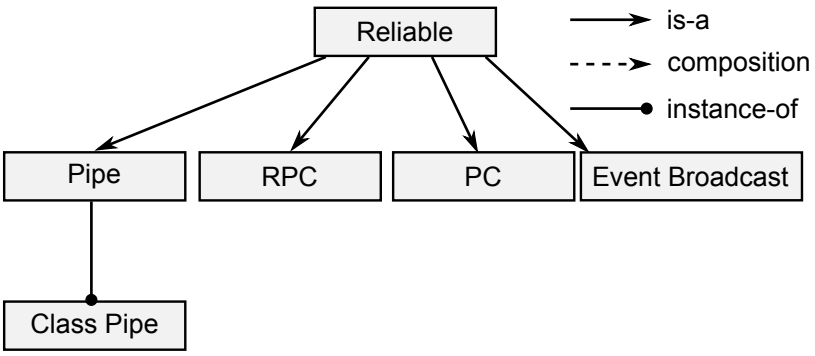


Figure 6.7: Connector classes taxonomy inspired by Hirsch et al. [1999]

$$\begin{aligned}
\text{RPC} &\leq \text{Knows Target} & \text{PC} &\leq \text{Knows Target} \\
\text{RPC} &\leq \text{Request Reply} & \text{PC} &\leq \text{Request Reply} \\
\text{RPC} &\leq \text{Synchronous} & \text{PC} &\leq \text{Synchronous} \\
\text{Pipe} &\leq \text{Flow Control} & \text{Pipe} &\leq \text{One Way} \\
\text{Event Broadcast} &\leq \text{Broadcast} & \text{Shared Data} &\leq \text{Broadcast} \\
\text{Pipe} &\leq \text{Streamed} & & \\
\text{Pipe} &\leq \text{Reliable} & \text{RPC} &\leq \text{Reliable} \\
\text{Event Broadcast} &\leq \text{Reliable} & \text{PC} &\leq \text{Reliable} \\
\text{RPC} &\leq \text{Typed} & \text{Event Broadcast} &\leq \text{Typed} \\
\text{PC} &\leq \text{Typed} & \text{Shared Data} &\leq \text{Typed}
\end{aligned}$$

It allows the definition of a combinator with type $\iota \to \iota$ that is a container building block $\mathfrak{b}_{\text{SharedMemory}}$ for shared memory connector:

$$\mathfrak{b}_{\text{SharedMemory}} : (\iota \to \iota) \cap \text{Shared Data}$$

Everywhere where a property "Typed and Broadcast" ($\text{Typed} \cap \text{Broadcast}$) for a connector is required by components, the building block $\mathfrak{b}_{\text{SharedMemory}}$ can be used, because $\text{Shared Data} \leq \text{Broadcast}$ and $\text{Shared Data} \leq \text{Typed}$ holds.

The connector classes by Hirsch et al. [1999] have been picked up, refined, and extended by Mehta et al. [2000] and more completely presented in [Taylor et al., 2010, pages 157ff].

The (atomic) connector types described here are:

- Procedure call (in Figure 6.8)

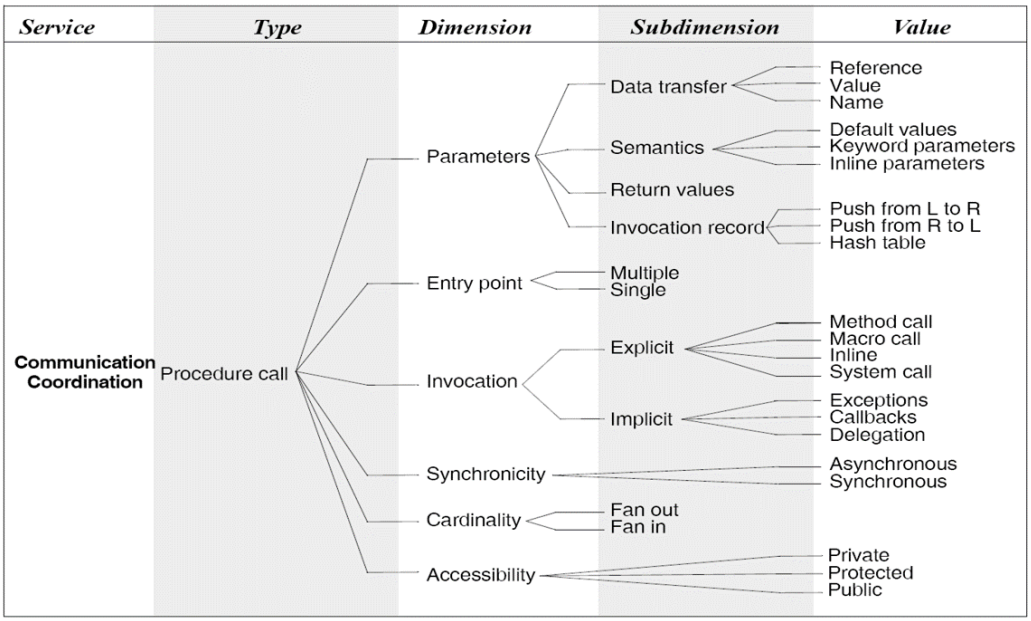


Figure 6.8: Example connector type Arbitrator from Taylor et al. [2010, pages 166ff]

- Data access
- Event
- Stream
- Linkage
- Distributor
- Arbitrator (in Figure 6.9)
- Adaptor

These taxonomies can be combined with a connector compatibility matrix in Figure 6.10 to form a complex practical taxonomy respectively subtyping extension for Combinatory Logic Connector Synthesis. A way of coding negations and exclusions for $\mathsf{BCL}_k(\cap, \leq)$ is presented in [Düdder et al., 2013a].

The composite connector types described in [Taylor et al., 2010] that can be composed from the atomic connector types are:

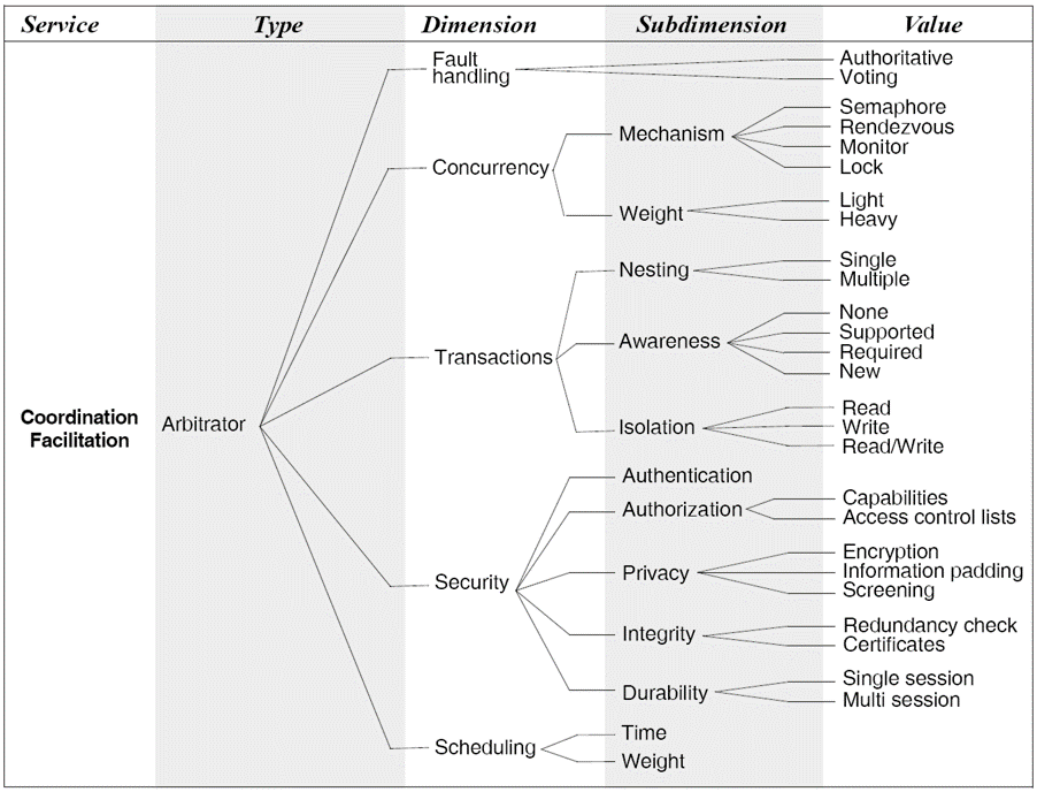


Figure 6.9: Example connector type Arbitrator from Taylor et al. [2010, pages 171f]

- Cloud/grid connectors (e.g., Globus[1])
- Procedure call
- Data access
- Stream
- Distributor
- Peer-to-peer connectors (e.g., Bittorrent)
- Arbitrator
- Client-server connectors
- Event-based connectors

[1]Project website: `http://toolkit.globus.org`.

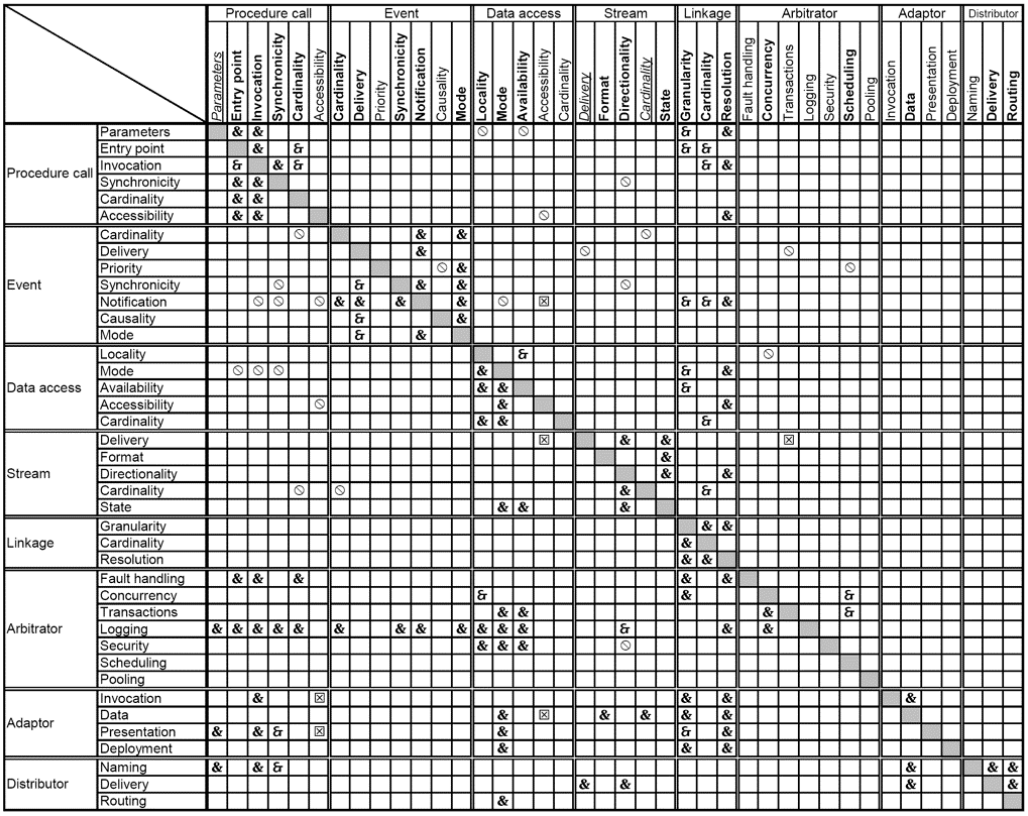


Figure 6.10: Connector compatibility matrix from Taylor et al. [2010, pages 178f]

We developed a tool, named ArchiType, to conduct experiments on realistic software architectures. ArchiType includes a taxonomy (coded as subtypes) that subsumes the taxonomy in [Taylor et al., 2010]. The building blocks have been developed for different target technologies like .NET or Java. The complete taxonomy is too large to be discussed here, therefore we provide an excerpt in Figure 6.11.This excerpt is focused on the concept of `Reliable Messaging`. `Object` is the root of all concepts in this connector taxonomy. Furthermore, concepts of `Communication Role` presented in Subsection 2.2.2 on page 28 and `Connector Type` have two sub concepts `Pipe` and `Linkage`, here. Furthermore the concept communication-`Protocol` with the relevant sub-concepts `HTTP`, `TCP/IP`, and `Mail` can be used as channels by `REST` (*representational state transfer* (REST) [Fielding, 2000]) and together with `Mail` as channels by WS-`SOAP`. A `Message Queue` is a kind of `Reliable Messaging` and `RabbitMQ` is an instance or product providing a `Message Queue` service. Note, that the instance `RabbitMQ`

is specified in the corresponding C&C type environment as a combinator $\texttt{RabbitMQ} : \texttt{MessageQueue} \cap \texttt{Linkage} \cap \texttt{Pipe} \cap \texttt{REST} \cap \texttt{SOAP}$.

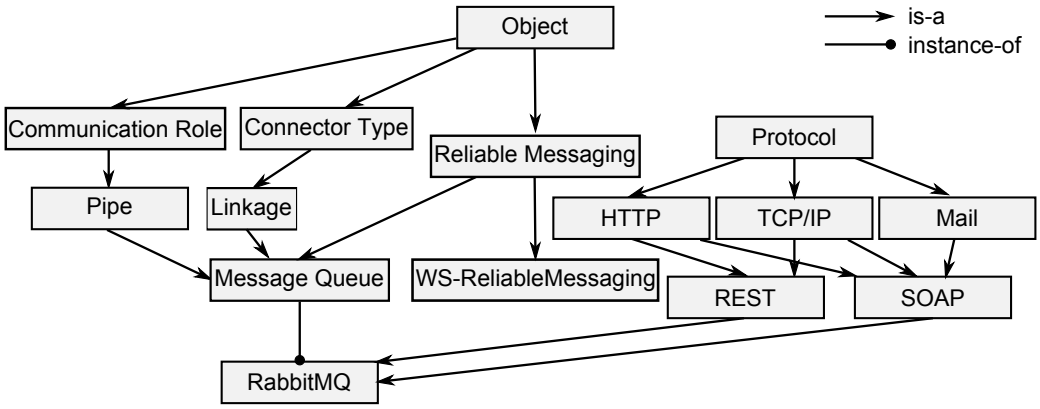


Figure 6.11: Excerpt of a connector taxonomy included in ArchiType

6.3 Combinatory Logic Connector Synthesis

For the definition of the Combinatory Logic Connector Synthesis, we need some more definitions for the method's steps.

6.3.1 Synthesis of Connector

In order to use type inhabitation, we have to *specify* a corresponding C&C type environment Γ_R as well as an atomic extension of the subtyping relation $\leq$ representing a taxonomy. Furthermore, we have to provide templates that are linked to the combinators in a C&C type environment and a type inhabitation question formulated as intersection type τ. This synthesis goal is given *schematically* where types $\iota_1, \ldots, \iota_l$ are instantiated by concrete interfaces.

Definition 6.3.1. ***(Connector synthesis goal)*** *A type* $\iota_1 \cap \rho_1 \to \ldots \to \iota_l \cap \rho_l \to \bigcap_{k \in K} \iota'_k \cap \rho$ *where* $m \geq 0$ *and* $l, |K| > 0$ *is called a* connector synthesis question.

The intuitive meaning of the connector synthesis goal is that we have two or more components that have a set of provided interfaces $\iota_1, \ldots, \iota_l$ and a set of required interfaces $\iota'_k \in K$. The connector to be synthesized has to connect the required interfaces of the components with their provided interfaces and

vice versa for the provided interfaces of the components. Note the structural similarity of a connector synthesis goal and a container building block.

We now have gathered all essential elements for using the type inhabitation $\Gamma \vdash ? : \tau$ for synthesizing a software connector description. Such a software connector description is an inhabitant e which is an applicative $\mathsf{BCL}_k(\cap, \leq)$ term defined in Section 3.3 on page 39 that satisfies $\Gamma \vdash ? : \tau$.

6.3.2 Generation of a Connector

The remaining step is the generation of a software connector by interpreting the inhabitant e. We interpret the inhabitant on its applicative term level corresponding to the following definition. The definition is aligned to a domain theory over interpretations.

Definition 6.3.2. *(**Generation of a connector**)*
An inhabitant e is interpreted as software connector by recursive application of cases in the interpretation function $[\![\cdot]\!]_\mathfrak{I}$ under a given interpretation $\mathfrak{I}$ defined by

1. *$e \equiv a$: if e has no arguments, $a : \bigcap_{I \in \mathcal{I}} I \cap \rho$, and ρ an intersection of properties, then $[\![e]\!]_\mathfrak{I}$ is connector named e that provides interfaces $I \in \mathcal{I}$.*

2. *If $e \equiv a(b_1, \ldots, b_n)$ and a is of a type not representing an atomic building block: Term e is an application of n arguments $b_1, \ldots, b_n$ with a, then the required interfaces of component $a$are connected to the corresponding provided interfaces of $[\![b_1]\!]_\mathfrak{I}, \ldots, [\![b_n]\!]_\mathfrak{I}$.*

Note that the cases of the definition above are excluding each other. The interpretation of an inhabitant e provided by $[\![e]\!]_\mathfrak{I}$ is abstract with respect to concrete objects like a software component and a container building block. We will use different interpretations to *generate* different objects. These objects will be concrete like UML2 components in a UML2 component diagram, program source code, configuration code, and so on. The program source code is generated by using the linked text templates of a combinator as its interpretation. In [Düdder et al., 2012] and more advanced in [Düdder et al., 2014b] we present an interpretation that generates Petri-Nets via graph expansion.Summarizing, the generation process can produce (nearly) arbitrary artifacts by an appropriate interpretation.

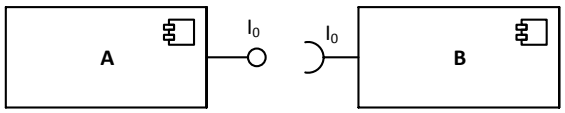


Figure 6.12: Components A and B

6.3.3 Combinatory Logic Connector Synthesis Method

This section and the previous Section 6.1 provide concise definitions for the Combinatory Logic Connector Synthesis method in Figure 2.2 on page 26 that is an essential contribution of this thesis.

Definition 6.3.3. *(**Combinatory Logic Connector Synthesis**)*
The Combinatory Logic Connector Synthesis method consists of the following steps executed sequentially:

1. Specifying *a C&C type repository and a taxonomy represented by an atomic extension of $\leq$ (by analyzing an architectural scenario).*
2. Synthesizing *compositions of building blocks by posing suitable connector synthesis goals.*
3. Generation *of objects (for example using code from the templates) by a suited interpretation function.*

We will say *synthesize and generate* and mean the sequential composition of specifying, synthesizing, and generation. This method can now be applied to synthesize[2] specific software connectors. In the following sections and chapters this method will be evaluated by conducting different experiments. But before we begin with that, we apply this method to a small example for demonstration.

6.3.4 Example for Combinatory Logic Connector Synthesis

We present a small example illustrating the Combinatory Logic Connector Synthesis method. We will come back to this small example in an experiment described later.

Example 6.6. *Consider the two components* A *and* B *depicted in Figure 6.12 that interact via interface I_0 which is required by* A *and provided by* B*. Both*

[2]It might be confusing that synthesis occurs in the methods name as well as in the name of a step in the method.

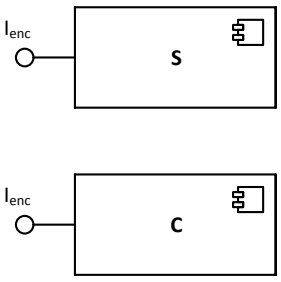


Figure 6.13: Components S and C

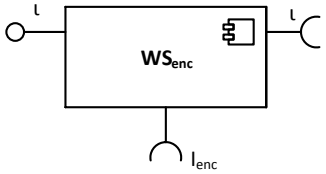


Figure 6.14: Building block $\mathtt{ws}_{enc}$

components reside in the same process and shall be distributed in a way that guarantees message security by encryption, i.e., we want to synthesize an I_0-connector that guarantees message security.

Assume a simple taxonomy that states sym and asym are encryption concepts (symmetric and asymmetric encryption, for instance in [Schneier, 1995]). Also assume a repository R containing two components, S and C depicted in Figure 6.13, providing functionalities for symmetric (by a shared secret) respectively asymmetric (by a certificate with public and private key pair) encryption. Both components provide the same interface I_{enc}. Furthermore, there is a container building block in the repository R that requires I_{enc} and is an I-connector that implements an interaction via a web service that guarantees message security by encryption.

We extend $\leq$ with the axioms $sym \leq enc$ and $asym \leq enc$. The repository Γ_R contains $(\mathsf{S} : I_{enc} \cap sym)$ and $(\mathsf{C} : I_{enc} \cap asym)$ (each combinator is semantically specified by an intersection stating what kind of encryption is provided) and the container building block $(\mathtt{ws}_{enc} : (I_{enc} \to (\iota \to \iota) \cap ws) \cap (\alpha \to \alpha))$ depicted in Figure 6.14. If $\mathtt{ws}_{enc}$ is given an interface of type I_{enc} it produces an I-connector for any I (by instantiating ι with I) by a web service ($\iota \to \iota$ is semantically specified with the type ws). Furthermore, $\mathtt{ws}_{enc}$ may transfer any property that α can be instantiated with to the resulting connector.

We synthesize the desired I_0-connector (i.e., an encrypted connection via web service) with connector synthesis goal $(I_0 \to I_0) \cap ws \cap enc$ by posing the

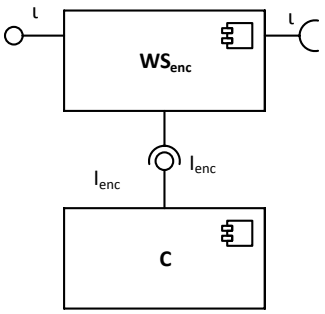


Figure 6.15: Resulting connector $\mathtt{ws}_{enc}(\mathsf{C})$

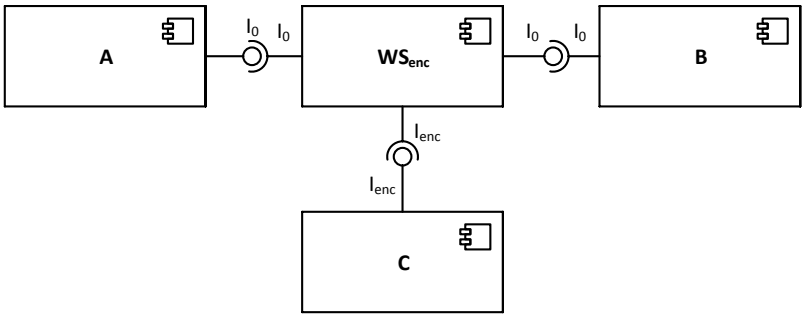


Figure 6.16: Resulting connector $\mathtt{ws}_{enc}(\mathsf{C})$ with components A and B

inhabitation question $\Gamma \vdash ? : (I_0 \to I_0) \cap ws \cap enc$*, and obtain two inhabitants* $\mathtt{ws}_{enc}(\mathsf{S})$ *and* $\mathtt{ws}_{enc}(\mathsf{C})$*. If, for example, we specifically want an* I_0*-connector with asymmetric encryption we pose the analog inhabitation question* $\Gamma \vdash ? : (I_0 \to I_0) \cap ws \cap asym$ *and only obtain* $\mathtt{ws}_{enc}(\mathsf{C})$ *in Figure 6.15.*

With a substitution $\iota \mapsto I_0$*, the resulting connector can be used to connect components* A *and* B *as shown in Figure 6.16.*

6.3.5 Synthesis of Behavior

In object-oriented software behavior and data are treated separately. The same holds for computation and interaction in C&C software architectures. Sometimes, interaction is more than mere data transfer and needs protocols as discussed in Section 2.2.2 on page 28. A way to specify protocols is using finite-state machines. Finite state machines and even ATMs can be simulated in $\mathsf{BCL}_k(\cap, \leq)$.

A simulation of a two counter machine[3] simulated by the type inhabitation in simple types $\lambda_{\to}$ is shown by Rehof [2013]. The type inhabitation in

[3]A two counter machine has two registers for counting integer numbers.

combinatory logic used here is even more expressive. This can be seen in the proof of the lower bound of type inhabitation in $\mathsf{BCL}_k(\cap, \leq)$. In [Düdder et al., 2012] the halting problem for $exp_{k+1}(n)$-space bounded ATM is shown to be reducible to relativized type inhabitation in $\mathsf{BCL}_k(\cap, \leq)$.

An advantage of employing $\mathsf{BCL}_k(\cap, \leq)$ for synthesis is the possibility to use higher-order typed combinators for composition. In C&C connector synthesis, data-adapters (or converters) map values from domain A into domain B. This data-adapter can be used in complex building blocks, for example as arguments such in $(\tau_A \to \tau_D) \to \iota$. Such higher-order typed combinators are used in the experiment "Detailed Broker Pattern Example" in Section 8.2 on page 170 as a façade-pattern adapting incompatible interfaces of clients and servers.

6.3.6 Designing C&C Type Repositories and Templates

Aldrich [2008] defines *architectural conformance* as: "A system conforms to its architecture if the architecture is a conservative abstraction of the run-time behavior of the system." This description is primarily focused on a system's behavior. Furthermore Aldrich [2008] states that "[...] conformance is a critical issue for *any* architecture." We can see that from an architect's point-of-view, architectural conformance is not only desired but also a critical requirement. Otherwise, the system's abstract architecture and the run-time system have different behavior.

This architectural conformance must also hold between the C&C type environment with linked templates and the synthesized *and* generated run-time system as well. Preserving the architectural conformance depends on three key assets of the Combinatory Logic Connector Synthesis method. First, it depends on the specification of a complete taxonomy. This completeness means that all relevant concepts and relations have to be specified. This is achievable by an iterative refinement of these specifications. Second, it also depends on the design of an appropriate C&C type environment *and* the proper specification with taxonomical concepts from the first step. Third, a conformant design and development of code templates with a balance between behavior (the runtime of the system) and its *data*. A more advanced but more complex synthesis approach with a preservation property is discussed in Section 7.6 on page 159. These three key assets must be specified correctly, aligned to a group of scenarios, in order to preserve architectural conformance.

For the construction of a C&C repository, a set of steps and transformations can be given.

The step for this design process is to determine a system's interconnection and interaction needs. For this, software interconnection models can be

helpful. The roles of the connectors can be analyzed and transferred into a classification for connector types. A classification using roles as a classifier is discussed in Section 2.2.2 on page 28. Summarizing, the discussion provides the following roles for a software connector: communication, coordination, conversion, and facilitation. In addition, Taylor et al. [2010] also suggest to determine the connector's appropriate types and its dimensions of interest. Exemplary dimensions and values, taxonomic ranks, for a special software connector are depicted in Figure 6.8 on page 131. Then, in the next step, appropriate values for each dimension should be chosen. Multi-types[4] described by Taylor et al. [2010], for example for composite connectors, can be spliced into compatible atomic connectors. For connector synthesis, the distinction between application-specific functionality (components) and application-independent interaction mechanisms (connectors), as discussed in Section 2.2.1 on page 26, offer an indication for distinguishing components from connectors.

During the design of building blocks, a set of transformations might be applied to the connectors in order to create composite building blocks. Spitznagel and Garlan [2001] propose a set of such transformations for connector construction. This set of transformations consists of the following elements:

1. **Data Transform**: "Data Transform changes the format of the data being exchanged in an interaction."

2. **Splice**: The splice transformation "[...] combines two binary connectors into a new binary connector".The splice operation can be generalized to n-ary connector splicing.

3. **Add a role**: "The transformation add a role adds a new interface (or "role") to an interaction to enable a new party to be involved".

4. **Sessionize**: changes connectionless protocols to session-oriented protocols and vice versa.

5. **Aggregate**: "[...] combines two or more connectors with a controller",where only one connector is active at a time.

These operations can be applied to connectors in order to create composite connectors.

An automatic model extraction algorithm could simplify the design phase for the Combinatory Logic Connector Synthesis method. For example, a type

[4]Such multi-types are software connectors that belong to different types of software connectors, e.g. a grid or a peer-to-peer connector.

system and a related model extraction algorithm are presented in [Lam and Rinard, 2003]. The type system in [Lam and Rinard, 2003] seems to be very expressive and whereas it is unclear if the corresponding inhabitation problem for this type system is decidable.

Remark 6.2. *Because of the complex inner structure and unique behavior that is needed for the application's business logic, a business component's possible reusability is drastically reduced. Applied to the components, the reduced reusability makes our methodology less effective. However, components encapsulating cross-cutting concerns of the software system represent an exception. We conjecture that these components providing services like security or logging, can also be synthesized and generated by extending our method to those components.*

General Development Guidelines

Some methods have been proven useful in our experiments and applications of (CL)S. The collection of development methods is not complete. Primarily, a good advice is the usage of logic programming methods and principles as presented in [Nilsson and Maluszynski, 1990]. The expressive power of logic programming has been surveyed in [Dantsin et al., 2001]. An investigation that is closer to the combinatory logic synthesis has been presented in [Miller et al., 1991] for uniform proofs as a foundation for logic programming.

(CL)S uses the relativized inhabitation in $\mathsf{BCL}_0(\cap, \leq)$ for synthesis. It means that type variables can only be substituted by intersections of type atoms. This might not be sufficient in all scenarios, for instance, where type variables have to be substituted by functions. A way to approximate substitutions in $\mathsf{BCL}_1(\cap, \leq)$ with $\mathsf{BCL}_0(\cap, \leq)$ substitution is to introduce fresh type variables. For example type variable α in $\mathsf{BCL}_1(\cap, \leq)$ can be instantiated with intersections of type atoms as in $\mathsf{BCL}_0(\cap, \leq)$ but also with terms of level 1 that are intersections of arrows ($\rightarrow$) with intersections of type atoms in argument and target position of the arrows. With type variables β_1, β_2, and σ, α can be approximated by the type

$$\sigma \cap \beta_1 \rightarrow \beta_2.$$

In application scenarios like in Combinatory Logic Connector Synthesis or in Petri-Net synthesis in [Düdder et al., 2012], a property coming from a set of properties has to be transported from a target position in an arrow type to the next inhabitation question. This can be encoded by an appropriate substitution of type variable α by the set of properties and the type

$$\tau \cap \alpha \rightarrow \tau' \cap \alpha.$$

Whenever a type is inhabited with a type $\tau \cap p$ with property p, the inhabitation question spawned for the argument will be $\tau \cap p$. The now transferred information on p can then be used for that inhabitation question.

Remark 6.3. *Note that this encoding differs from an encoding that replaces the substitution of the property (type variable α) by subtyping. Assuming $p \leq p_1, \ldots, p \leq p_n$ with properties p_i for $i \in \{1, \ldots, n\}$, an encoding using subtyping would have the type*

$$\tau \cap p \to \tau \cap p.$$

From that, the ($\leq$)-rule allows to conclude

$$\tau \cap p_i \to \tau \cap p_j$$

for $i, j \in \{1, \ldots, n\}$ whereas the substitution with α only instantiates as

$$\tau \cap p_i \to \tau \cap p_j$$

with $i = j \in \{1, \ldots, n\}$. This means that the case $i \neq j$ cannot be simulated by using a single variable. By introducing a second variable for the first or second occurrence of α, the resulting substitutions can be instantiated with arbitrary $i, j \in \{1, \ldots, n\}$. By using subtyping, we lose the information on the correlation of instantiations of used type variables of a path.

An important feature of intersection types is the possibility to encode *finite functions* via their function tables. The encoding of a monadic function f over a domain D can be encoded by

$$\bigcap_{x \in D} x \to f(x).$$

This encoding can easily be generalized to polyadic functions and is used in Definition 6.2.3 for the connectors in Combinatory Logic Connector Synthesis and more explicitly in the setup of the experiment presented in Section 4.1.5 on page 66.

The encoding of finite function tables in intersection types can be used to implement the transition function of *finite state machines*. Then, the inhabitation can be used to simulate the execution of a finite state machine in combination with an appropriate encoding of finite state machine's states, input and output. A two-counter machine encoding is presented using (for simple type $\lambda_{\to}$) in [Rehof and Urzyczyn, 2012]. A more complex encoding for $exp_{k+1}(n)$-space bounded ATMs is used in the (K+2)-EXPTIME hardness proof in [Düdder et al., 2012].

Finite existential quantification, $\exists x.P$, and finite universal quantification, $\forall x.P$, of a formula P can be mimicked in $\mathsf{BCL}_k(\cap, \leq)$ by applying $\cap$ to all valuations of x for $\exists$ or by applying $\cap$ to all valuations of x in contravariant position of $\rightarrow$ for $\forall$.

Last of all, a recursive but strong normalizing combinator x can be given an intersection type $\tau \rightarrow \tau \cap \tau'$ where $\tau \rightarrow \tau$ represents the recursive part and the type τ' represents the result of the termination. Such an intersection type is a valid alternative to the usage of ω to type strong normalizing combinators. By adding a designated type constant that prevents recursive inhabitation, we can eliminate undesired cyclic inhabitants. Such a designated type constant is the type constant f that is used in Section 4.1.5 on page 66.

6.4 Related Work

Various logics have also been used in the context of synthesis within software architecture to obtain end-to-end realizations from abstract architecture descriptions to code where the focus varies between specification of entire architectures and specification and synthesis of connectors.

Broadly speaking, the approaches may be distinguished in at least three dimensions. First, manual approaches to composition can be contrasted with automatic approaches, the former aiming for verification of composed architectures by using powerful, sometimes undecidable, specification logics whereas the latter rely on simpler, decidable logics. Second, some approaches view components as black-boxes that only expose their interfaces whereas others also describe the behavior of the components, for example, by protocols. Last, the approaches differ in what is synthesized, ranging from the synthesis of abstract architecture elements such as ADLs or views, for example, to concrete code.

A classical approach to use higher order logic (HOL) for the specification of software architectures is presented in [Allen and Garlan, 1994] which is based on the ideas described by Garlan and Shaw [1993]. The specification language Z has been proposed for architectural specifications in [Coombes and McDermid, 1991]. Moriconi and Qian [1994] propose modal logic to specify software architectures and their manual composition but the focus lies in verifying intensional correctness of a composed architecture. SAT-solvers are proposed by Maoz et al. [2013] for structural architectural synthesis using Alloy developed by Jackson [2002].

Maoz et al. [2013] use also propositional logic to specify C&C architectures. One particular difference of the work in [Maoz et al., 2013] compared to our work is that our type theoretic approach also models the functional dependency

between a connector's provided and required interfaces. Furthermore, (CL)S based on $\mathsf{BCL}_0(\cap, \leq)$ presents a more expressive language which is also reflected in the complexity of the corresponding decision problem (NP versus 2-EXPTIME). Ports are a part of the model presented by Maoz et al. [2013]. Such a port can be represented in our model by a designated covariant type constructor containing the interfaces that are pooled by such a port. In this thesis, ports are not taken explicitly into account, because ports are no first-class citizens in C&C architectures as defined by, for instance, Taylor et al. [2010]. Nevertheless, we generate one individual port for each interface of a component in a UML2 component diagram, because a components-ports-interfaces connection is compulsory for valid UML2 component diagrams.

de Alfaro and Henzinger [2001] present interface automata that are used by Inverardi and Tivoli [2013] and Autili et al. [2010] to describe the behavior of components. Connectors whose main purpose is to adapt varying behaviors are synthesized. Yellin and Strom [1997] synthesizes component adapters that comply with protocol specifications. Fiadeiro et al. [2003] provide an ADL agnostic and independent formalism to specify the semantics of architectural connectors by category theory. This specification is intended for the system's description.

Penix [1999] proposes a first-order logic specification of architectural components that is used to retrieve components from a library based on matching conditions. The construction of exogenous composite connectors via a hierarchy is presented by Lau et al. [2007]. In this work, the composition is done manually and components are specified by sequences of logical atomic predicates. A current work by Bennaceur et al. [2013] on the automated synthesis of mediators [Buschmann et al., 1996, pages 410ff] describes a formal method and also presents a tool using an ontological reasoner in combination with behavioral specifications to synthesize such mediators.

Most of these specification proposals aim to architectural documentation and/or verification. Prevalently, even approaches for composing architectures by specifications are manual approaches that guarantee architectural conformance according to those specifications.

The idea of composite connectors like in our method and the ideas presented by Spitznagel and Garlan [2001] and Julien and Perry [2008] have also been proposed as composable context-aware architectural connectors as presented by Julien and Perry [2008] as abstract objects. The author's interest in composable connectors is motivated by their usage for expressing self-adaptive software architectures. Interestingly, the authors note that: "To our knowledge, no one has ever before considered composing connectors; although different uses of connectors have been distinguished." [Julien and Perry, 2008, page 1].

The work presented here is also related to the work that has been done in the domain of semantic web services (*web service modeling ontology* (WSMO) and *ontology web language* (OWL-S) and related reasoning capabilities), e.g. Rao and Su [2004], the use of *artificial intelligence* (AI) planning techniques for the selection of proper compositions, e.g. by Bertoli et al. [2010] and Kazhamiakin et al. [2013], or the use of genetic algorithms by Aversano et al. [2006] to determine suitable compositions.

Waters and Abowd [1999] provide a good overview of general synthesis processes for software architectures. The main focus in this overview is the integration of different architectural perspectives for legacy software architectures.

Chapter 7

ArchiType

In order to demonstrate the applicability of the Combinatory Logic Connector Synthesis method presented in Section 6.1, we implemented ArchiType as an extension to Microsoft Visual Studio Ultimate 2013[1] (VS2013) which includes an architecture extension supporting UML2. ArchiType also uses (CL)S as a synthesis service. In principle, we want to generate connector descriptions in an arbitrary ADL, however, in this work we restrict ourselves to the usage of UML2-component diagrams [Object Management Group (OMG), 2005] as an ADL.

We adapted UML2-component diagrams to include several stereotypes capturing the types describing components and interfaces and their properties. ArchiType allows the initiation of connector synthesis by selecting specific interfaces to be connected in actual UML2-component diagrams. The system proposes synthesized solutions in the form of compositions of building blocks, and the architect (user) chooses an eligible solution. ArchiType automatically generates a description of the synthesized connector within the diagram depicted in Figure 8.8 on page 182 below) as a UML2-packaging component [Object Management Group (OMG), 2005] from the chosen composition. This process may be repeated for all interfaces for which connectors are needed. This yields a UML2-component diagram that is enriched with the generated connector descriptions.

Primarily, ArchiType is technology and language agnostic with respect to generation targets, e.g. Java. It uses a database as data-store and can be extended or be reconfigured easily. The current version allows different technologies like Microsoft WCF and the open-source communication framework ServiceStack.[2] We will focus on WCF because the generated code is easier to

[1] `http://www.microsoft.com/visualstudio/eng/products/visual-studio-ultimate-2013`

[2] The project's website is `http://www.servicestack.net/`

present.

The enriched UML2-component diagram is then interpreted in order to generate compilable code for all synthesized connectors from the underlying templates of the used building blocks. For this code generation, ArchiType uses the Microsoft T4 Text Templating engine. Each building block in the repository has to be provided with a set of suitable T4-templates. The synthesis specification is then translated in an ArchiType Composition Language script. Controlled by the ArchiType Composition Language script, we use the T4-engine to generate C#-source code as well as XML-configurations for connectors. An extended approach with *staged composition synthesis* (SCS), an extension for (CL)S, for the generation of ArchiType Composition Language scripts as well as implementation type correctness are discussed. Following up, implementation details of (CL)S and notable features are presented.

We will now discuss these topics in more detail in the following sections.

7.1 UML2 Extension

For the purpose of connector synthesis, type information must be added to the UML2-components. UML2 has two categories to extend existing UML2-diagrams.

The first category are meta-model changes like using the *meta-object facility* (MOF) defined by the Object Management Group (OMG) [2011].[3] MOF is standard model-driven approach defined by the *object management group* (OMG). MOF defines a complex meta-model for UML2 that allows extension or modification of the UML2 model. A disadvantage of MOF is its support in industrial tools, because it modifies UML2 in a non-trivial way, e.g. by adding semantic, that complicates the integration of MOF into these tools.

The second category are UML2 modifications by existing constructs within the standard of UML2. In this category, the extensibility mechanisms are tags, constraints (for instance using the *object constraint language* (OCL), or stereotypes.

The residual third category are stereotypes that are a mechanism allowing to extend the vocabulary of UML2 by creating new model elements. These model elements are reusing existing model elements and can have supplementary, specific properties. Graphically, a stereotype is notated by guillemets ($\ll \cdot \gg$) enclosing the stereotypes name ($\cdot$). Stereotypes are commonly supported in today's industrial software engineering tools (by way of example, Microsoft Visual Studio or IBM Rational).

[3] OMG's MOF webpage: `http://www.omg.org/mof/`

We defined four stereotypes containing a property named `Type`. This property is of data type string and contains $\mathsf{BCL}_k(\cap, \leq)$ type information in the form described in Section 5.3 on page 110 and in Appendix A.1 on page 199. The defined stereotypes of ArchiType are:

- ≪`ArchiType component`≫: This stereotype is used to add a property named "types" to UML2 components.
- ≪`ArchiType connector`≫: This stereotype is used to add a property named types to UML2 connectors. This connector also contains a string property called `WCF-Configuration`. This property can be used to link a UML2 connector to specific WCF-configuration files.
- ≪`ArchiType part`≫: This stereotype is used to add a property named types to UML2 parts. UML2 parts are a general group of different model elements in UML2. One of these parts is the UML2-interface and thus UML2-interfaces also contain the property `Type`.
- ≪`ArchiType port`≫: This stereotype is used to add a property types to UML2 ports.

These stereotypes are integrated in the ArchiType extension as a file describing a profile. This profile is an XML-document containing a collection of the stereotypes, meta-classes, and property type definitions. These stereotypes contain a meta-class moniker definition that specifies in which contexts this stereotype can be employed, for example within UML2-component diagrams or within UML2-activity diagrams.

Such a profile has to be activated manually for the UML2-component diagram. Then, a new property appears for marked objects in the property view of objects called `Type`. This property can be edited and the $\mathsf{BCL}_k(\cap, \leq)$ type can be placed as string there.

More profiles containing different stereotypes are integrated in Visual Studio 2013. Such stereotypes allow storing information, for instance of source code file information or namespaces, in the properties of UML2-model elements. This information can also be used by ArchiType provided that such information is actually available.

7.2 User Interface

ArchiType is fully integrated in the Architecture Modeler of Microsoft Visual Studio 2013 Ultimate. Thus, the first step in ArchiType's usage is to open or to create a UML2 component model project. This project can be created in

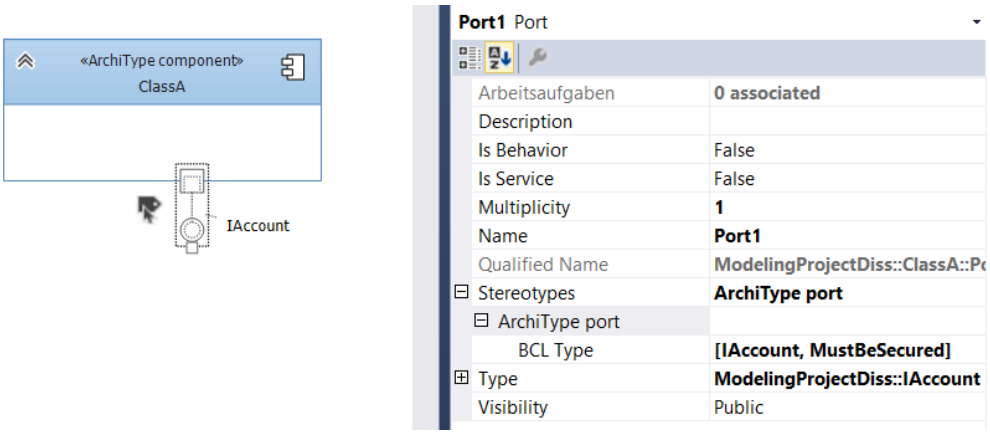


Figure 7.1: Specification of ArchiType specific stereotypes for UML2 elements

a new project solution or an existing project solution that already contains other projects. In the UML2-component diagram view, existing classes, for instance written in C#, can be added by dragging their classes onto the UML2-component diagram view. For these classes, the corresponding stereotypes as described in Section 7.1 must be applied to the according component and its child model-elements. A UML2-component has UML2-ports containing UML2-interfaces. Then, the needed UML2-model elements must be specified by types. This specification is shown in Figure 7.1.

After marking two or more interfaces in the UML2-component diagram, a context menu entry on the UML2-component diagram initiates the synthesis process with a web service call to a (CL)S-server. After computing the results, the user can select one out of these synthesized solutions in a dialog. A selected solution is now interpreted as UML2-connector and a corresponding ArchiType-component model for the UML2-connector is created. This step can be repeated for more interfaces and, thus, more UML2-components for UML2-connectors are generated.

Another context menu entry starts adjacently the code generation process. The user has to select a target technology and language that ArchiType offers. For this target technology and language specific code is generated. After the process stopped, a total generation time or a list of error messages is displayed.

7.3 Synthesizing Connectors in UML2

The selected target technology and language influences the automatic construction of the specialized C&C type environments. The software connector

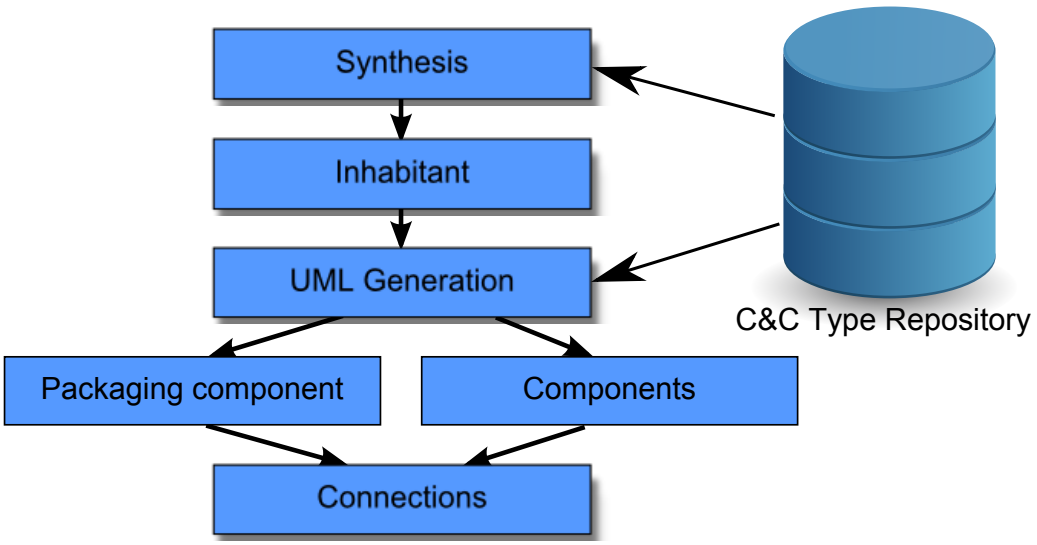


Figure 7.2: Generation process from inhabitant to UML2 connector

is generated from a synthesized inhabitant of the specialized C&C type environment. This inhabitant is an applicative term in which its components represent building blocks as described in Section 6.2.3 on page 122 to Section 6.2.3 on page 123. The generation from such an inhabitant to a packaging component is done by interpreting the components of the applicative term's AST in top-down traversal. Figure 7.2 shows the process. The process starts with an existing UML2-component diagram containing at least two selected UML2-components to be connected. The following steps generate a UML2-component diagram that is enriched with a generated connector resp. its UML2-packaging component.

A container building block is represented by a UML2-packing component with the top-level *main* component in a packaging component. A UML2-packaging component is then connected to two or more UML2-components that shall be connected by a synthesized connector. The connections of a UML2-packaging component are constructed with respect to linked $\mathsf{BCL}_k(\cap, \leq)$ type information as well as provided and required interfaces. Also, ports of the UML2-packaging component are connected via delegate connectors to the main component of the UML2-packaging component. Here, an argument position in a term resp. position of a child in an AST is used for a correct correlation between two components to connect.

Every other component in the inhabitant is interpreted as a corresponding UML2-component. The interfaces of components are connected via connectors. A component of the inhabitant's required interface must be connected to a

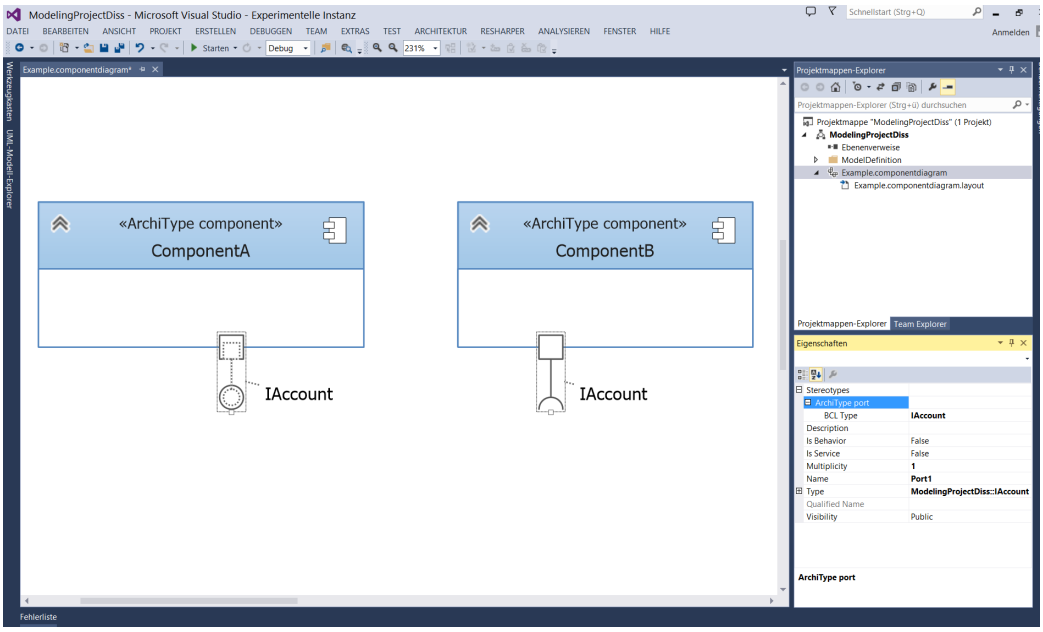


Figure 7.3: ArchiType screenshot before generation

provided interface of its child or argument components.

Following, a corresponding port for each interface is inserted between a component and an associated interface. Furthermore, every generated UML2-component is given a correct $\mathsf{BCL}_k(\cap, \leq)$ type information in a corresponding stereotype property depending on a C&C type environment in a repository. Such a type information is later used for a code generation process described in the next section.

7.4 Code Generation

Code generation is the process to translate connector model elements (in the form of UML2-packaging components) in a given synthesized UML2-component diagram into source code fragments. Generated source code fragments and existing source code, which is modeled by a UML2-component diagram, have to be compiled into an executable program. This process is a part of the model-driven engineering approach. The code generation process is depicted in Figure 7.5.

The depicted process uses an indirection via an intermediate language to control the generation process. Such an indirection is needed to bridge the gap between an abstract synthesized connector depicted in UML2 without any code reference with a very specific format of source code. The gap is missing

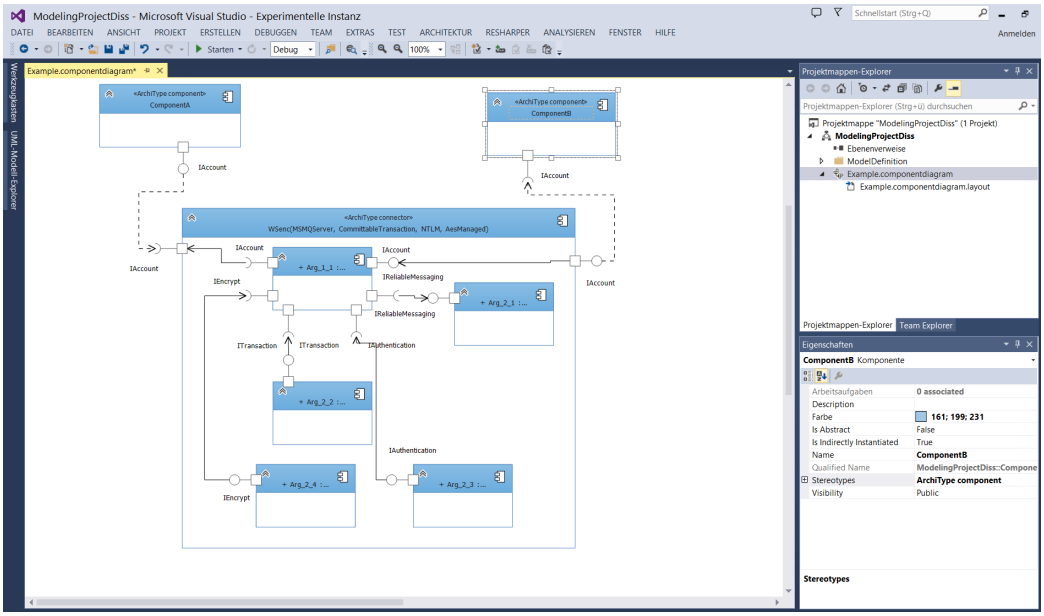


Figure 7.4: ArchiType screenshot after generation

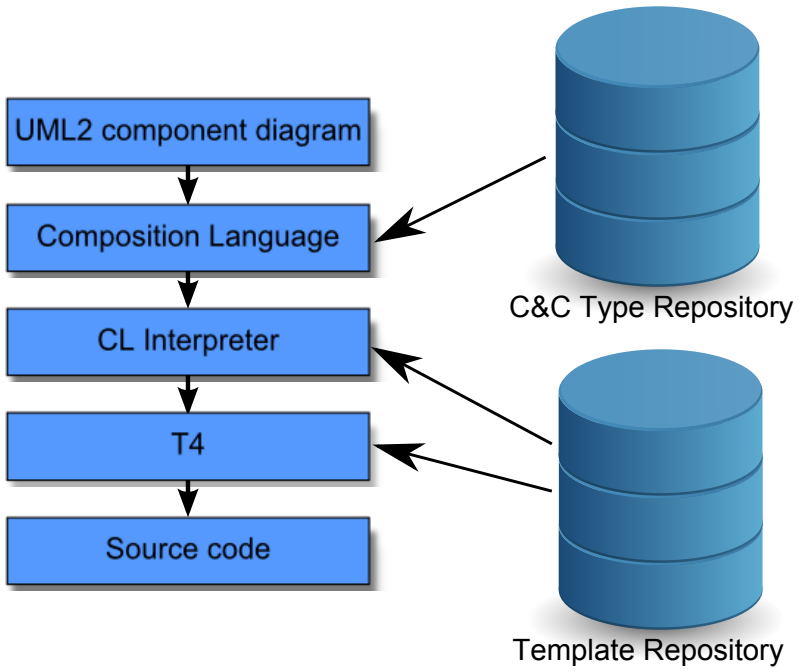


Figure 7.5: Generation process from UML2 to source code

information like directory or file names, or additional uses of software tools (for example for generating WSDL code from existing interface specifications). The intermediate language is called ArchiType composition language.

Every building block in the repository is linked to a set of template names. Such templates are used to generate script code. The script language will be discussed in the following section. The template's selection is determined by the chosen target technology and language resulting in different ArchiType composition language files depending on the technology and language. This variability allows for a flexibility and extensibility of ArchiType.

The generation starts with a UML2 package component implementing the connector and traverses to its children. For every component that is reached by a traversal, an entry in the repository is used to insert a set of template names into the script file. Note that such a traversal is alike to traversing an inhabitant from which a UML2-component diagram is enriched. Container building blocks may contain additional information depending on concrete technology associated with a building block.

7.5 ArchiType Composition Language

The ArchiType Composition Language is a textual DSL. The purpose of the ArchiType Composition Language and its interpreter is to:

1. **Control** the code generation process in compliance with the dependencies existing between the code templates.

2. **Provide** values for variables that are used in the code templates.

3. **Trigger** the Microsoft T4 engine to compile and execute source code templates in T4 code in a closure operation in order to generate concrete target source code.

4. **Evaluate** operations, e.g. concatenating strings.

5. **Handle** errors either from the interpreter (file not found) or from the T4 engine.

6. **Generate** multiple files. A software system contains a lot of source code files. A generator must be able to generate multiple files. For a contract-based client-server connection, source code for a client proxy, a server stub, a contract, communication codes (for example WS-SOAP or REST), and a lot of configuration files as well as triggering a lot of tools supporting the generation are necessary.

7. **Comply** with existing interdependencies between different template files as specified by the semantics added to the C&C type environment.

The composition language file is a text file that contains three sections. These sections have different purposes:

1. The first section is a declarative part, where a collection of template names is declared. The declaration format is function-like. Assume we have a template named T and this template takes three other templates as its arguments a_1, a_2, and a_3, then the declaration looks like:

$$T(a_1, a_2, a_3)$$

2. The second section is used to initialize string variables with values. A generic template file gets specific values via such variables. In order to distinguish template names and variables, variables have to begin with `$`. The section is also used by ArchiType to apply concrete values, for instance stemming from the UML2 component diagram to variables for a code generation in text templates.

3. The third section contains composition commands in an imperative script form. The script language is a sequence of commands including functional applications. A special binary infix operator `->>` has two arguments. The first argument is a command (template name) and the second argument is a string or variable containing a file name. The file name will later be used to write the result of a command into a file designated by the file name. It allows the generation of multiple files during an interpretation of the script. An example line in such a command sequence is:

```
T(A, B(C,D), E)->>''path/file1.ext''
```

A simple example for an ArchiType Composition Language code file is contained in the appendix in Section A.2 on page 201.

The ArchiType Composition Language interpreter expects that every template name in a script part has a corresponding template file with the same name. It parses an input file and checks if all declarations are correct. Afterwards, it initializes all variables with data from the second section. The second section forms a blackboard architectural pattern as a central data provider for the generation process. In the next phase, it picks the first command and interprets its arguments. The results of the arguments of the command then are passed as values to the first command. Command and

arguments are linked to templates via their name. The interpretation of commands and arguments is made by a call to a controller mapping argument names from a specification to variable names in a T4 template. Then, the ArchiType's T4 controller loads a template text and assigns this template text to a T4 processor. A T4 processor compiles and executes the template as executable and returns its return value to the ArchiType's T4 controller. If a first command is an argument of `->>`, then a return value is written to `->>`'s second argument interpreted as a file name. After that, a second command is interpreted. The interpreter stops, if an error is raised or the last command in a script has been processed.

The Microsoft *text template transformation toolkit* (T4) is a template based text generation framework included in Visual Studio. The T4 generates arbitrary text files and is therefore agnostic to target languages. T4 accepts a custom template format which can contain .NET code and string literals in it. String literals and .NET code are separated by two special symbols `<#` and `#>`. The .NET code contains meta-code for a code generation, is compiled, and executed. The result is embedded into a resulting text that contains a mixture of executed result and string literal parts.

Example 7.1. *A simple T4 text template is shown below:*

Listing 7.1: Example T4 template file

```
1  <#@ template debug="true" hostSpecific="true" #>
2  <#@ output extension=".cs" #>
3  <#@ Assembly Name="System.Core" #>
4  <#@ import namespace="System" #>
5  <#@ import namespace="System.IO" #>
6  <#@ import namespace="System.Collections" #>
7  <#@ import namespace="System.Collections.Generic" #>
8  <#@ parameter name="className" type="System.String" #>
9  <#
10   // this is meta code in C# that is executed
11    string Greeting = "Hello";
12 #>
13 // This is the output code from your template
14 namespace MyNameSpace{
15   class <#=className#>{
16     static void main (string[] args){
17       System.Console.WriteLine("<#= Greeting #>, the time is now:
             <#= System.DateTime.Now.ToString() #>");
18     }
```

```
19    }
20  }
21
22  <#
23    // Insert any template procedures here
24    // this is also meta code in C# that is executed
25    void foo(){}
26  #>
```

In this example template a C# file is produced that contains a namespace and a class declaration with a static void method **`main`**. *The static method is called if this program is compiled to an executable with console output. The method* **`main`** *outputs a string with time information. The content and the effect of the listing is:*

- ***Lines 1 to 8****: Are meta-template codes for configuring the T4-processor. Such a meta-template code is encapsulated by* **`<#@`** *and* **`#>`**.

- ***Line 1****: The* **`template`** *directive is used to set compiler specific options, like compiler options or culture variables used for specific cultural system settings (for example the user's language). If the option* **`hostspecific`** *is set to* **`true`**, *then a variable* **`Host`** *can be used. The variable* **`Host`** *is a connection to the T4 processor host which is in this case the* ArchiType*'s processor host. If the* **`debug`** *attribute is* **`true`**, *then information for the debugger is provided that enables the debugger to identify a position in the template where a break or an exception occurred more accurately. Analogously, such information is as well used by* ArchiType *to present warning or error information to the user.*

- ***Line 2****: The* **`output`** *directive is used to set the file extension of a file that is used for output. The flag can be used to override the setting that is made with the binary operator* `->>` *in the* ArchiType *composition language.*

- ***Line 3****: The* **`assembly`** *directive loads an assembly that allows a template code to use types defined in a template meta-code.*

- ***Lines 4 to 7****: The* **`import`** *directive allows referring to elements in another namespace without providing a fully-qualified name. The directive corresponds to the* **`using`** *directive in C#.*

- ***Line 8****: The* **`parameter`** *directive is used to set parameters externally to provided values. A data* **`type`** *must be provided for a parameter. A*

value of a parameter is set by the ArchiType *T4 processor and depends on all evaluated values of corresponding arguments of a template in the* ArchiType*'s composition language.*

- ***Lines 9 to 12****: The lines contain meta-code in C# encapsulated by* `<#` *and* `#>`*. Such a code is* compiled *and* executed *by the T4 processor. In line 11 the variable* `Greeting` *of type string is set to the value* `Hello`*.*
- ***Line 13 to 21****: These lines are interpreted mostly as string literal and are inserted in a generated file unchanged as program code in C#.*
- ***Line 15****: The class declaration is made for a class where the substituted parameter* `className` *is used.*
- ***Line 17****: In the meta code* `<#= Greeting #>`*, the variable* `Greeting` *is used in an execution of the program and a result replaces the corresponding section. In* `<#= System.DateTime.Now.ToString() #>` *a call to the class method* `now` *of class* `System.DateTime` *is made and the method's result is converted to a string replacing the original meta-code.*
- ***Line 22-26****: These lines are an example of declaring a void method in meta code that can be used during an execution of the T4 processor.*

The generated output file by ArchiType*'s T4 processor of such a template file with the parameter* `className` *set to* `MyGeneratedClass` *is:*[4]

Listing 7.2: Output of example T4 template file 7.1

```
1 using System;
2 // This is the output code from your template
3 namespace MyNameSpace{
4   class MyGeneratedClass{
5     static void main (string[] args){
6       System.Console.WriteLine("Hello, the time is now: 12:39 on
            12.01.2014");
7     }
8   }
9 }
```

A more complex file that is used for experiments in Chapter 8 in Section 8.4 on page 179 can be found in the Appendix in Section A.3 on page 202. All experiments in Chapter 8 have been conducted with templates in T4 and the ArchiType composition language.

[4]Under the assumption that `foo()` did not produce any output.

7.6 Staged Composition Synthesis

In the presented combinatory logic synthesis, synthesis provides no distinction between composition and program code. Such a distinction is made implicit by adding conceptual information to used combinators. However, the distinction between a type τ and a code of type τ can be made more explicit. The idea is called *staged composition synthesis* (SCS) presented in [Düdder et al., 2014a] and in more detail in a technical report [Düdder et al., 2013b]. In SCS the intersection type system is extended by a modal type constructor $\Box$. An intersection type $\Box\tau$ is interpreted as a *code of type* τ. The modal type constructor allows to separate two different and individual stages of synthesis. Analogously, the ArchiType Composition Language interpreter separates meta-computation in T4 meta-code and native computation in the target system. An appropriate type inhabitation algorithm like Algorithm 4.1 on page 53 is presented for SCS in [Düdder et al., 2014a]. An optimized implementation of the inhabitation algorithm for SCS as presented in Chapter 4 is used for program synthesis. Figure 7.6 depicts a brief overview of the different levels of computation of SCS.

	Languages	Notions of computation	
CL	Combinatory Logic	Composition-time	$\mathscr{C} \vdash ? : \Box(\mathtt{R} \cap Fh \cap ms)$
		Inhabitation	$(\mathbf{tmp}\ @\ (\mathbf{Tr}\ @\ \mathbf{O}))\ \Diamond\ \mathtt{cl2fh}$
L2	Compositional Metalanguage	Composition-time	$@ \triangleq \lambda F : \Box(\alpha \to \beta).\lambda X : \Box\alpha.$ letbox $f : \alpha \to \beta = F$ in letbox $x : \alpha = X$ in box $f(x)$
		Reduction	$(\mathbf{tmp}\ @\ (\mathbf{Tr}\ @\ \mathbf{O}))\ \Diamond\ \mathtt{cl2fh} \longmapsto^*$ box(let x : R = tmp (Tr 0) in x∗(9 div 5) + 32)
L1	Native Language	Run-time	
		Execution	let x : R = tmp (Tr 0) in x∗(9 div 5) + 32

Figure 7.6: Three levels of computation in SCS

The work on SCS is inspired by the $\lambda_e^{\Box\rightarrow}$ calculus presented by Davies and Pfenning [2001]. The operational semantics is the reduction relation $\longmapsto$ (and its reflexive transitive closure $\longmapsto^*$) defined for $\lambda_e^{\Box\rightarrow}$ in [Davies and Pfenning, 2001]. The code generation is implemented by a β-reduction and a letbox-reduction rule which is called $\Box\beta$ in [Davies and Pfenning, 2001]. The reduction $\longmapsto^*$ is responsible for the meta-computation (code generation) which results in a well-typed native language program, for example in the functional language ML. The well-typedness of the native language program is guaranteed in both works [Düdder et al., 2014a] and [Davies and Pfenning, 2001] by a conservative extension theorem. A similar reduction in SCS is presented and implemented as an extension to (CL)S that is used to conduct experiments on different scenarios.

Especially one additional optimization for SCS has been considered. This optimization poses a restriction on the substitutions that have to be generated. Substitution instances must contain exactly one single native type, because a native program must be well-typed and a common native language does only support simple types instead of intersection types. The requirement of containing one single native type instead of multiple native types reduces the possible number of substitutions.

The SCS approach is closely related to the approach with the ArchiType compositional language presented above in the previous section. The ArchiType compositional language is interpreted in a way that after an interpretation a well-typed program is generated. A similar well-typedness as in SCS cannot be guaranteed. On the other hand, the ArchiType compositional language with T4 templates presents an industrial tool that can generate multiple files using an abstract, high-level language like C#, whereas the SCS tool uses a functional language close to the λ-calculus. The functional language is, in turn, not easy to comprehend and to apply to complex scenarios.

The question arises, whether SCS would be an appropriate synthesis method for synthesizing software connectors? But it turns out that SCS is too restricted to be used as synthesis method for software connectors in the Combinatory Logic Connector Synthesis method. In the type-encoding presented in Section 6, software connectors are represented by a type $\tau_{\mathsf{c}} \equiv \bigcap_{\mathcal{P}' \subseteq \mathcal{P}} \sigma_{\mathcal{P}'} \cap \bigcap_{p \in P_{\mathsf{c}}} p$ according to Definition 6.2.5 on page 122. In this encoding, arbitrary top-level intersections of interface types $\sigma_{\mathcal{P}'}$ out of $\mathcal{P}' \subseteq \mathcal{P}$ might occur. Interface types $\sigma_{\mathcal{P}'}$ are primus inter pares without precedence and (semantically) meaningful distinction in software architecture. In SCS lingo, such interface types might be identified as native types in SCS. But as defined in [Düdder et al., 2014a, page 6], SCS only allows the occurrence of one single native type within an intersection type. This feature is in conflict with the type encoding presented here and hinders a usage of SCS in the

synthesis scenario of software connectors in the Combinatory Logic Connector Synthesis method. Even introducing a single dummy native type, e.g. a computational type `module`, would not solve the discussed problem so far and, in fact, it would reduce the effectiveness of SCS since all *semantic* information is located in the meta-computation. The efficiency of SCS results from a rule-based entanglement of two separate (staged) dimensions of specification. The first dimension is formed by a specification in a native language and a corresponding logic. Whereas in an orthogonal second dimension, meta-computation is specified by semantic concepts. Nevertheless, using a single native type constant collapses the first dimension to a singleton and only allows variability in the second dimension with minimal restrictions originating from the entanglement.

An experimental approach using a single designated native type `module` is adopted. Thereby, the SCS extension of (CL)S with a two-level type system and a distinction of computation phases can be used by ArchiType to generate ArchiType composition language files. Generated ArchiType composition language files are then interpreted by the ArchiType composition language interpreter that generates native code program files using underlying T4 templates.

Partial evaluation of programs by Jones et al. [1993] and [Jones, 1996] is also related to SCS in a broader sense. In partial evaluation input programs are optimized by specialization, e.g. to speed-up the program execution while guaranteeing the preservation of program behavior. Partial evaluation also distinguishes two phases between program transformation, by residualizing a program, and its execution in an untyped setting.

7.6.1 Implementation Type Correctness

In Section 6, implementation type correctness is preserved in the encoding of software connectors as types, in the well-typedness of building blocks, and in the implementation details of building blocks occurring in a synthesis solution.

Implementation type correctness for software connectors is equal to satisfaction of all may-connect properties in Definition 6.2.4 on page 121 for interfaces occurring in an inhabitant, because interface types are the only implementation types occurring in a connector type $\tau_{\mathsf{c}} \equiv \bigcap_{\mathcal{P}' \subseteq \mathcal{P}} \sigma_{\mathcal{P}'} \cap \bigcap_{p \in P_{\mathsf{c}}} p$ with $\sigma_{\mathcal{P}'} \equiv \tau_1 \to \ldots \to \tau_k \to \tau_{\mathcal{P}'}$ according to Definition 6.2.5 on page 122. Both conditions occurring in Definition 6.2.4 are argued separately:

- *Condition 1* of may-connect is satisfied because a building block occurs in an inhabitant, because by construction of a C&C type environment there must have been an argument i building block providing an interface τ_i

with $i \in \{1, \ldots, k\}$ that the current building block requires. Otherwise, the current building block respectively its provided interface would not have been inhabited.

- *Condition 2* is also satisfied because of the encoding of connectors as types and an inhabitant exists in which all required interfaces of building blocks are satisfied by corresponding provided interfaces by other building blocks. Subset $\subseteq$ in condition 2 corresponds directly to the implicit ($\cap$E) rule (cf. Section 3.3.1 on page 40) that is derivable using the ($\leq$) rule together with subtyping rules for $\leq$ in Section 3.2.2 on page 36.

Well-typedness of building blocks remains a task of the designer of a C&C type environment. But similar to the work of Haack et al. [2002], implementations of semantic types are not (formally) checked against their specification. This trade-off allows to hide and abstract complex implementation details and to refer to concise features via abstract semantic concepts represented as symbols instead.

Implementation type correctness for building block implementations is preserved by the fact that underlying templates associated to a building block are closed with respect to templates of other building blocks and only communicate via label/name sharing. Only container building blocks have a designated context in which child building blocks can interfere by code fragments under control of the container building block. Therefore container building blocks must be designed in a way such that isolation of its child building blocks is respected as well as the wiring of required and provided interfaces respectively their source code pendants is made in a sensitive manner. Especially, container building blocks need a thoughtful and purposeful design to guarantee (implementation) type correctness on a source-code level.

7.7 Implementation

ArchiType is implemented in C# as an extension package for Visual Studio 2013 Ultimate and uses the (CL)S service presented in the end of the last chapter. Figure 7.7 depicts a UML2 object diagram for ArchiType. Grey objects are part of the Microsoft Visual Studio 2013 Ultimate and are used by ArchiType. Arrows in this diagram represent directed dependencies.

Technically, ArchiType uses Microsoft's *managed extensibility framework* (MEF) to integrate itself into Visual Studio 2013. MEF is a container framework for extensible applications that is used by Visual Studio 2013. Specific commands and dialogs are used for user-interaction. A SQL Server instance is

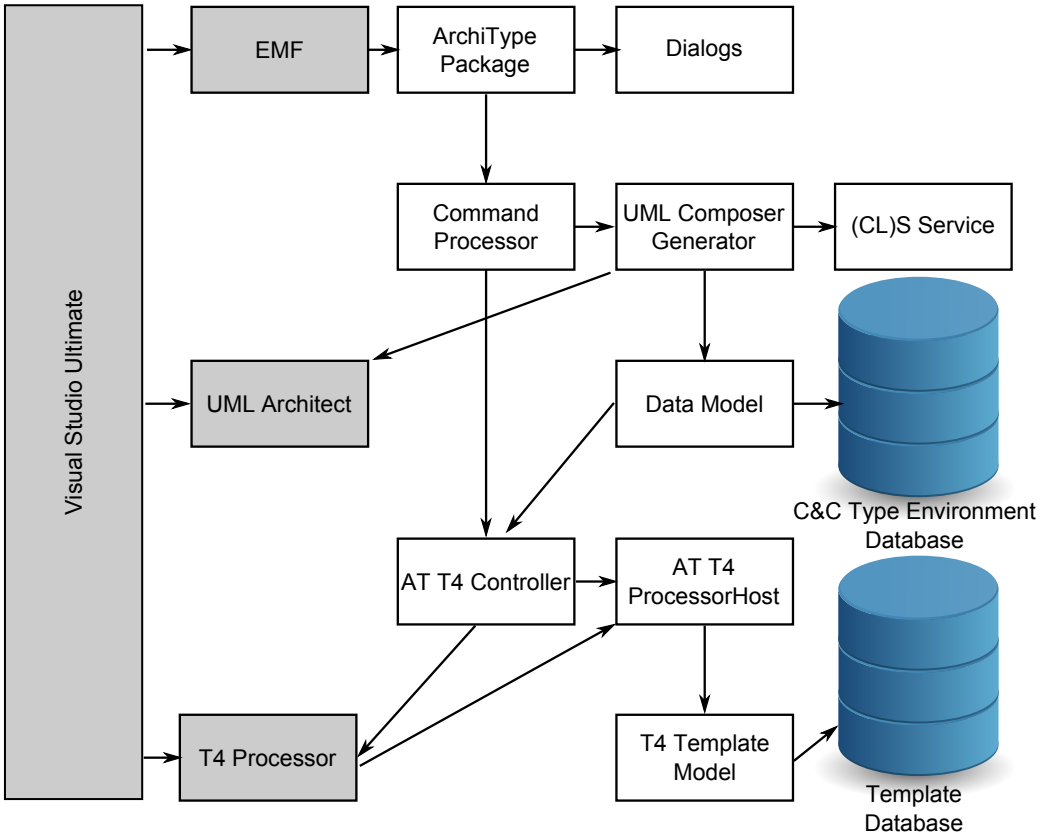


Figure 7.7: UML2 object diagram for ArchiType

used via the *entity framework* (EF) framework as a repository by ArchiType for persisting the C&C type environment as well as the T4 text templates. The EF framework is an object-relational mapper for relational data that is usable by domain-specific objects. The database also stores data for different target technologies and languages (as template variants). Using this implementation, experiments are conducted with T4 on bigger scenarios presented in Chapter 8 in Section 8.4 on page 179. There, a code generation for approximately 4000 lines of C# and related configuration code took less than 30 seconds. The figures suggest that the presented code generation process seems to be slow. After analyzing the results, the reasons and most time consuming operations are:

1. The repeated *compilation* and *execution* for every T4 template takes most of the dedicated computation time. If ArchiType is developed more in the direction of an industrial tool, then the generation process can be expedited by the usage of precompiled T4 templates. Such T4

templates have been precompiled, cached by ArchiType, and reused on demand preventing repetitive compilations of the same T4 template.

2. The repeated *initialization* of the .NET application domain that is used to form a closure for each T4 template generation (for instance to prevent parameter clashes with a similar name caused by other possibly concurrent executions).

It is important to mention that code generation by T4 is not a time-critical operation, if ArchiType is used as a prototyping tool. In some of the conducted experiments in the next chapter, ArchiType is also used to generate deployment code and configurations for generated connectors and applications. For example, in the e-Commerce example in Section 8.4 on page 179 configurations for a Microsoft Internet Information Server hosting the eCommerce solution were also generated to speed up the experiments and to reduce errors.

7.8 Related Work

Using ArchJava by Aldrich et al. [2002] and Alloy by Jackson [2002], Bagheri [2011] presents an end-to-end approach to automatic synthesis that is similar to our approach in that we also generate ADL (UML2) and source code in Java. In ArchJava [Aldrich, 2008] data types can be used to preserve communication integrity which is a valuable property of architectural conformance. ArchJava and related approaches differ from Combinatory Logic Connector Synthesis because ArchJava has no explicit concept for connectors. Connectors can only be approximated by using ArchJava's links and ports. Furthermore, ArchJava does not provide a synthesis method.

Model-driven engineering tools are also related to ArchiType. Such tools are using model-to-model- and model-to-code-transformer even in sequence forming so called model-pipelines. Model-driven engineering tools also contain wizards to aid an architect and/or developer to create or modify models. Such wizards are very rigid and do not offer the flexibility of the synthesis/generation method proposed in the thesis.

Chapter 8

Applications and Experiments

We apply the Combinatory Logic Connector Synthesis method to concrete software architectures and evaluate its applicability by means of conducted experiments. We will conduct the experiments in a rigid structure: An experiment starts with an initial question to be answered by the experiment. An experiment is designed whose results are solving the posed question. An appropriate environment for the experiment is setup in which the designed experiment is conducted. Afterwards, experimental results are presented, analyzed, and discussed.

We begin with a motivating example that develops the example presented in Chapter 6 in Subsection 6.2.4 on page 125 further. This motivating example will primarily focus on the design of a C&C type environment and synthesis with relativized type inhabitation. In contrast to the motivating example, a more complex example is discussed with a strong focus on code generation aspects. In this example, a many-to-many client-server application is transformed by introducing a supplementary broker pattern in addition with specific software connectors that are constructed on-demand.

Two more examples will provide more insight on practical, concrete applications and scalability of connector synthesis. The systems will be a mid-scaled (38 250 LLOC) and large-scaled (225 763 LLOC) application. Both scenarios have in common that both assume an application involving several components that resides on a single machine (monolithic architecture). This monolithic architecture shall be adapted into a distributed, network-based architecture for which software connectors (with various properties) have to be synthesized and generated. The reference system the experiments were executed on is the same as the one described in Section 4.1.5 on page 66 and is described in more detail in the appendix in Chapter B on page 207.

Subsequently to the last experiment, we will present a collective analysis of the experimental results and discuss the findings with regard to applicability,

adaptability, costs and effort, usability, and limitations of the Combinatory Logic Connector Synthesis method with respect to the experimental application scenarios.

We begin with two synthetic examples to exemplify the Combinatory Logic Connector Synthesis method and related aspects in concrete code generation.

8.1 Secure Connectors

The intention of this experiment is to demonstrate the applicability of our method in an in-vitro scenario that allows us to explain different features of our method in more detail. We will present in detail how the Combinatory Logic Connector Synthesis method bridges the gap from an abstract, type-based specification to a concrete, executable program source code.

8.1.1 Setup

We performed experiments with ArchiType on the example discussed in Section 6.2. We used Microsoft *Windows Communication Foundation* (WCF)[1] to implement a messaging system. WCF is a communication framework based on the ABC-paradigm, realizing communication by providing an address, a behavior(-specification), and a contract. We opted for WCF, because it is extensible, fits to our compositional approach WCF, and allows behavioral extensions.

8.1.2 Execution of the Experiment

We consider two components A and B depicted in Figure 6.12 that interact via interface I_0 which is required by A and provided by B. Both components reside in the same process and shall be distributed in such a way that guarantees message security by encryption, i.e., we want to synthesize an I_0-connector that guarantees message security.

We implemented A and B as well as I_0 in C# with a simple mock-up functionality (37 *logical lines of code* (LLOC). We also implemented Γ_R as described in Section 6.2 on page 120 by linking each combinator to a set of T4-templates for generating configurations. The combinator $\texttt{ws}_{enc}$ is provided with three underlying T4-templates realizing a SOAP-web service. The first template uses the C#-declaration of interface I_0 as a contract to generate a *web service description language* (WSDL)-file and a characteristic XML-Schema. From these files a C#-proxy for the implementation of I_0

[1] `https://www.rabbitmq.com/`

and a WCF-client configuration are derived defining address, binding, and behavior of the connector. The second template generates a WCF-server hosting the implementation of B, exposing I_0 as a service. The server needs a WCF-server configuration which is generated from the third template. The combinators S and C are provided with one T4-template each which specify symmetric respectively asymmetric encryption within the two WCF-configurations (client and server) by shared secrets respectively by certificates utilizing a *public key infrastructure* (PKI).

8.1.3 Results

Choosing one of the inhabitants $\texttt{ws}_{enc}(\mathsf{S})$ and $\texttt{ws}_{enc}(\mathsf{C})$, ArchiType generated a UML2-component diagram[2] and a complete client-server application distributing A and B with generated proxy and stub. For this application 257 LLOC were generated within 8.1s. It should be noted that solving the inhabitation question only made up for a fraction (162ms) of the execution time. The repeated compilation and also repeated initialization of the T4-engine consumed most of the execution time. In this motivating example the ratio between LLOC of existing code (implementations of A, B, and I_0) versus generated code is approximately 7.0. Such high ratios cannot be expected for real scenarios, for example, such as discussed in Section 8.4. However, for such scenarios the software connector codes are usually very complex.

The program code in Figure 8.1 was automatically generated as a partial C# class extending `ClassA` with method `Initialize` to create a proxy with `interfaceB` that redirects method calls from `ClassA` to `classB`. The server hosting the service `B` with `interfaceB` is situated on the local machine listening on port `8080`. Line 5 contains a WCF configuration not as an XML-configuration but as a program code. Both configurations have the identical effect with respect to a messaging configuration of WCF.

The program code in Figure 8.2 contains an excerpt from the program code of the server hosting the service `classB` with interface `interfaceB`. The service host is instantiated in line 1 and opens a host in line 11. In line 12 a diagnostic output is made to a console for every endpoint generated. The endpoints are defined in a WCF-configuration. In this experiment we configured SOAP Web Service and NetTCP endpoints for the service. Analogously to the previous listing, such a configuration can also be implemented using program code, but would have the disadvantage that a recompilation of the program code is required after changing the configuration.

Even this motivating example requires a lot interlinked program code,

[2] Akin to the one presented in Figure 8.8 below.

```
1  public partial class ClassA
2  {
3    private void Initialize()
4    {
5      _refToB = new
           InterfaceBClient("DefaultBinding_InterfaceB_InterfaceB",
6                  "http://localhost:8080/ICSESimpleExample.ClassB/");
7    }
8  }
```

Figure 8.1: Generated code for `classA` in secure connectors scenario

```
1  using (var hostSimpleExample.ClassB = new
       ServiceHost(typeof(SimpleExample.ClassB)))
2  {
3    try
4    {
5    // Server settings section
6    //TODO: Change connection settings
7      // Open the ServiceHost to start listening for messages. Since
8      // no endpoints are explicitly configured, the runtime will create
9      // one endpoint per base address for each service contract
10     // implemented by the service.
11   hostSimpleExample.ClassB.Open();
12   foreach(var uri in hostSimpleExample.ClassB.BaseAddresses)
13     Console.WriteLine("The service SimpleExample.ClassB is ready at
           {0}", uri);
14   //...
15   } catch(...) { ... }
16 }
```

Figure 8.2: Generated code for service host `Server` in secure connectors scenario

therefore a manual implementation of connectors is error-prone and, in general, should be avoided. We only had to manually adapt 2 out of the 257 generated LLOC to specify the address and the binding of the generated service. Note, that there was no necessity to adapt the original source code in order to integrate it with the generated code.

8.1.4 Analysis and Discussion

We extended the motivating example slightly by adding further building blocks offering more services which resulted in a large increase in the combinatorics (number of possible compositions of combinators). ArchiType resolves and thus hides these combinatorics from the software architect.

We added four authentication concepts (and combinators corresponding to those concepts), for example, authentication by Kerberos , for instance described in [Schneier, 1995], or an issued token to the taxonomy. These concepts can be combined with the aforementioned encryption concepts. We also added a corresponding combinator, stating that encryption respectively authentication can be disabled, i.e., such combinators allow opting for the synthesis of software connectors for messaging connections that do not require any security or authentication.

Next, we introduced a container building block realizing a messaging connection by a TCP-pipe as opposed to a web service-connection, e.g. using WS-SOAP. On an orthogonal scale, we added for each of these building blocks a corresponding combinator of the same type implementing the same functionality not by a configuration but by a direct C#-implementation. This way (also allowing hybrid solutions involving code and configurations or servers exposing web service- and TCP-pipe-transport), we synthesized 240 different I_0-connectors for distributing A and B if no properties were required of the connector. The computation of the corresponding inhabitation question took 84ms.

By requiring additional properties of the desired I_0-connector expressed by additional concepts in the taxonomy, we pruned the number of solutions. For example, if we ask for a TCP-pipe connection with authentication using a NT LAN Manager and symmetric encryption we obtain 8 connectors that can be chosen from as needed.[3] Theoretical results by Rehof and Urzyczyn [2011a] show that it is always possible to specify the repository in such a way that for a given inhabitation question a unique solution can be found (see for Proposition 4.2.2 on page 87). Using ArchiType, we generated I_0-connectors for A and B for more than 50 out of the 240 synthesized combinations and

[3]The 8 variants originate from the fact that we allow mixing code and configuration.

tested them. On average this generation took approximately 16.5s. This indicates that inhabitation is not necessarily the determining factor with respect to time consumption.

8.2 Detailed Broker Pattern Example

The intention of the broker example is also to demonstrate the applicability of our method in a controlled and more realistic scenario that allows us to discuss the power of the specification using intersection types and the simultaneous fulfillment of multiple orthogonal requirements. Additionally, we want to synthesize and generate behavior of connector that are conform to specified (communication-)protocols.

8.2.1 Setup

In this synthetic example, we will focus on certain details of code generation. This example corresponds to a classical example that is described and motivated by Buschmann et al. [1996]. The scenario is a software system consisting of 6 clients and 4 servers as depicted in Figure 8.3. Each server provides a separate service. The clients are using services provided by the servers. Every client needs for its tasks all services provided by the servers. This leads to a classical $n \times m$ problem for n clients and m servers and the architecture needs $n \times m$ connections between clients and servers. As discussed in [Buschmann et al., 1996, pages 237ff], such a software system has many disadvantages and problems as well as it hinders the evolution of the software system. Buschmann et al. [1996] proposes a solution to this problem that introduces a broker pattern into the software system. The broker decouples clients and servers by an indirection over itself. The broker in Figure 8.4 becomes a central relay for all communications between clients and servers. Such a software architecture design change does not completely solve the problem for a software architect. After introducing the central broker, one problem still remains, how do client and server interact with each other through software connectors? Even tough we reduced the amount of needed software connectors from $n \times m$ to $n + m$, but these $n + m$ connectors still have to be implemented. Furthermore, in realistic scenarios, we cannot expect that we have to implement a homogeneous set of software connectors. Each software connectors might have to conform to individual requirements of each client and server. Even in this example, 10 software connectors have to be synthesized and generated.

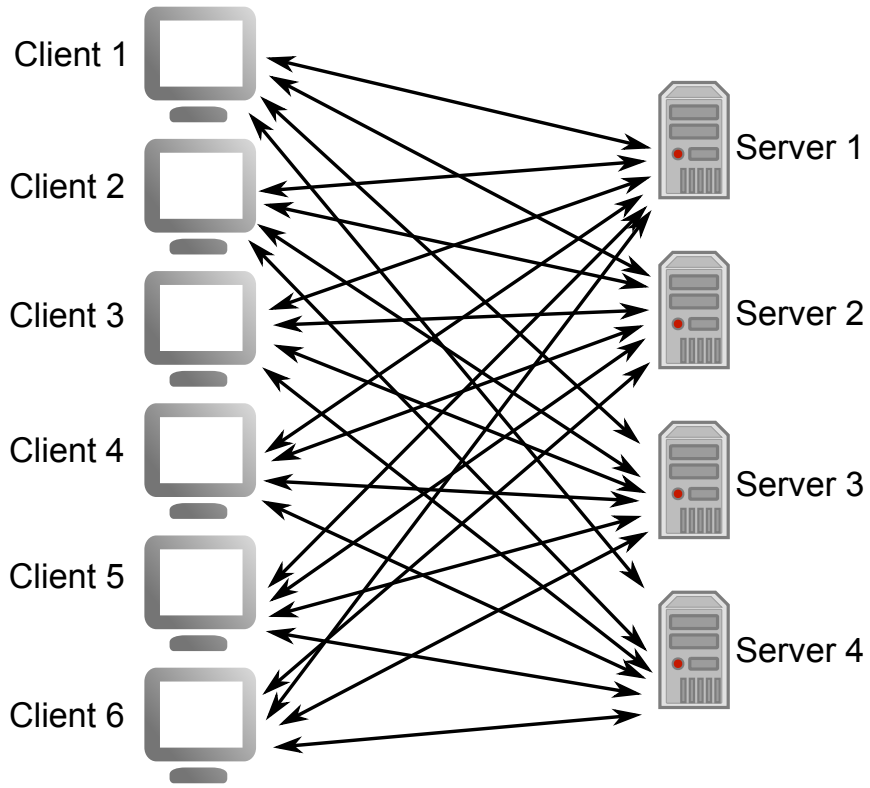


Figure 8.3: Many-to-many client server application

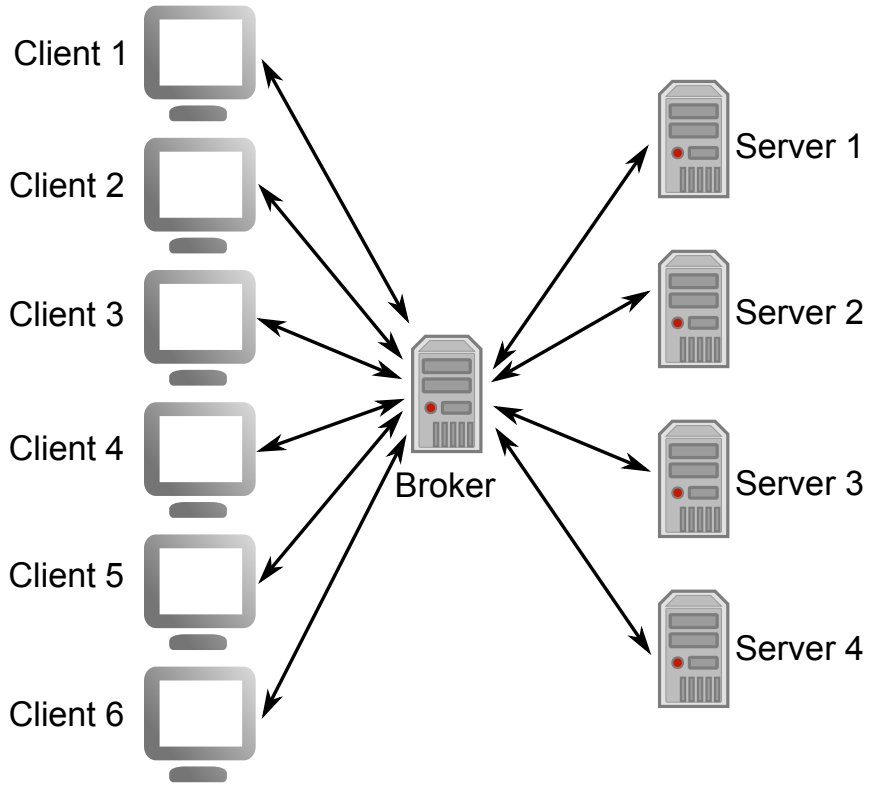


Figure 8.4: Many-to-many client server application with broker pattern [Buschmann et al., 1996, pages 237ff]

We use ArchiType with an appropriate C&C type environment and templates to synthesize these software connectors automatically. We now describe the setup of the experimental system and additional requirements which are as follows.

For every server, we have a server application exposing service interfaces $I_{S1}, \ldots, I_{S4}$ with different semantic properties in P.

- Server 1 uses a SOAP web service (I_{S1}) to provide its service.
- Server 2 also uses a SOAP web service (I_{S2}) to provide its service and in addition the web service requires authentication.
- Server 3 uses a REST web service (I_{S3}) to provide its service.
- Server 4 uses a NetTCP web service (I_{S4}) to provide its service. The web service also provides a supplemental secure connection channel via asymmetric encryption using a certificate (implying an existing PKI-infrastructure).

8.2.2 Execution

In order to increase the experiment's complexity, we introduce a further not unrealistic requirement for a single client. Client 1 needs a façade (pattern) [Buschmann et al., 1996, pages 294ff] for interface I'_{S1} to connect to server 1's (or broker's) service interface I_{S1}. An additional adapter building block is needed for synthesizing this façade. This adapter building block uses higher-order typed combinators that provide data-adaptation / -conversion functionality. Every client is a console application using all four services interfaces $I_{S1}, \ldots, I_{S4}$ provided by the servers.

The broker requires that all its messaging [Buschmann et al., 1996, pages 221ff] satisfies all the individual requirements demanded by the servers. We present the capability of our approach by transforming the initial synthesis question for $n + m$ connectors to a synthesis question for synthesizing only a single connector encapsulating a broker pattern inside. The pattern also includes the behavior to relay messages to the correct receiver (client or server). Then, we have to generate program codes and configurations for $n + m$ software connectors and the broker. This broker software connector is now a n-to-m software connector.

The original code for clients and servers is 116 LLOC resp. 169 LLOC of C# and 318 LLOC resp. 236 LLOC of WCF-configuration. The code contains many repetitions for the $n \times m$ connections.

Every software connector in the system was generated from its interface declaration used as a contract. All contracts have been compiled into appropriate WCF-configurations. The broker's container building block provided interfaces $I_{S1}, \ldots, I_{S4}$ and also its WCF-configuration was generated. The services with interfaces $I_{S1}, \ldots, I_{S4}$ of the broker relay the requests to the correct server *and* also the responses to the correct client. The broker uses asynchronous completion token [Buschmann et al., 1996, pages 268ff] to correlate communication requests and responses.

8.2.3 Results, Analysis, and Discussion

An appropriate C&C type environment has been designed to capture and reflect the demanded (semantic) requirements of the Combinatory Logic Connector Synthesis method described in Chapter 6. An appropriate UML2-component diagram has been sketched and complementary stereotypes have been assigned scenario-adequate types. After choosing one particular solution out of 4096, ArchiType generated the necessary code consisting of 993 LLOC in less than 7 seconds. The 4096 solutions resulted from the 4^6 variations based on the four different semantic properties for the communication channels.

The server addresses had to be manually modified. Then, the servers and the clients could be executed without errors according to the specified setting.

Even though the broker's container building block must be specified for all variants of n and m connectors for clients and servers, the T4 template is made smart enough to allow different cardinalities n and m of connections. Therefore, all container building blocks of the container are linked to a single set of T4 template files. Here, a set is needed, because a server consists of multiple individual source code files, C# source code files, WCF-configurations, and a deployment description for easier testing.

The example and the related experiment demonstrate that ArchiType and our method is adaptable to such application scenarios as well as powerful enough that software connectors with *arbitrary* complexity[4] can be synthesized and generated. Furthermore it demonstrates that the behavior of a connector to execute protocols can be hidden by encapsulating the behavior in an abstract specification of a corresponding building block and in associated templates implementing the behavior.

8.3 Enterprise Resource Planning Scenario

The underlying question for ERP experiment is how effective and with how many manual adaptions can the Combinatory Logic Connector Synthesis method be applied to a realistic, mid-scaled software application that is not streamlined for the Combinatory Logic Connector Synthesis method.

8.3.1 Setup

We apply the method to a first real-world software system to evaluate its applicability and scalability. We chose an open-source *enterprise resource planning* (ERP) solution, XERP.NET[5] as the test object, because it is large and complex enough for meaningful experiments, and its code is freely accessible. Furthermore, it is suitable for ArchiType since it uses the .NET-framework and also is based on *plain old C# Objects* (POCO). We prefer POCO systems for these experiments, because POCO systems are more controllable and easier analyzable, e.g. no dependency injection is used. It also means that the application is not developed using component-based development. XERP.NET as an application is realistic, because it is commercially operated in small- and mid-sized enterprises.

[4] As long as the software connectors are composable.

[5] `http://xerpdotnet.codeplex.com/`

8.3.2 Execution

XERP.NET is modular system consisting of 137 subsystems/components with 691 classes with 38 250 LLOC altogether. The software architecture of XERP.NET is layered. Again, the goal is to distribute the monolithic object-oriented architecture/implementation of the system and to synthesize the connectors that are necessary to distribute the components into a network-based architecture. We defined the partition points for the system on the premise to minimize impacts on functional and non-functional system properties. The architecture of XERP.NET is such that all interactions between components are one-to-one. Thus, for the chosen partition points we only have to synthesize I-connectors replacing the procedure calls between the involved components. As discussed in Section 6.2, more complex connectors can be specified and synthesized if the underlying system requires interaction between several interfaces of various components (multicast), for example in the detailed broker pattern example in Section 8.2 on page 170.

In our scenario the complexity arises from the fact that we allow for a great variability in the connectors: we use a comprehensive taxonomy classifying 21 concepts describing connectors (our taxonomy is inspired by Hirsch et al. [1999] and Mehta et al. [2000]). This taxonomy proved to be well-suited to control connector synthesis in the context of XERP.NET. For example, we added combinators and templates for transactions, reliable messaging via a message queue server (for instance provided by RabbitMQ[6]), application caches like memcached[7]/redis[8]/CouchDB[9], data services like OData[10], a composable security model, and many more. Usually, it is to be expected that the architect specifying a suitable repository for connector synthesis for a specific scenario possesses a comprehensive domain knowledge. Often, this will allow for the exclusion of certain elements of the repository. Ultimately, the repository Γ used in the following contained 24 building blocks from the variety of building blocks mentioned above.

Since all affected components of XERP.NET do not exhibit interfaces, we first refactored those components by automatically extracting their interfaces. We then proceeded as follows. We imported the components into an appropriate UML2-component diagram. Using dependency analysis for this diagram we determined two partition points in the server system for which we wanted to synthesize connectors. As discussed, we chose these points such

[6] `https://www.rabbitmq.com/`
[7] `http://memcached.org`
[8] `http://redis.io/`
[9] `http://couchdb.apache.org`
[10] `http://www.odata.org/`

that the number of dependencies was small, in order to reduce the possible impact on functional and non-functional system properties. Furthermore, if necessary, this makes manual adaptation of the connectors easier. For the involved components we applied appropriate stereotypes that define the types of their interfaces as defined in Section 7. For each partition point we performed synthesis of a connector. Depending on the connector specification there were several solutions. We chose a best-suited solution, which was then included as a UML2-packaging component representing the connector into the UML2-component diagram depicted in the ArchiType screenshot in Figure 8.8. Once all connectors were included in the diagram we started the generation of the different connectors.

Figures 8.5 and 8.6 contain visualizations of the subsystem dependencies of XERP.NET created with an architecture tool in JetBrains Resharper 8.1, a refactoring tool for Visual Studio 2013. The visualized subsystems contain in mean more than 280 LLOC and nearly 5 classes per subsystem. The dependency view depicted here was used to identify partition points that are congenial to the condition of minimal dependency.

We consider one out of these partition points in more detail. The chosen partition point is the component `XERP.Server.DAL.MenuSecurityDAL`. This component delivers meta-data for the security system of XERP.NET by using an entity framework class from a database. The intention to place a partition point here is that XERP.NET is connected to the security system hosted in another environment via a newly synthesized connector. The component `XERP.Server.DAL.MenuSecurityDAL` is used by 7 other components. We chose a broker pattern as described in the previous example to connect all 7 service clients with the partitioned service residing in a newly generated host next to an existing database server hosting the master data of XERP.NET.

The following modifications to the original source code of XERP.NET and manual operations had to be made:

1. Extracting service interface definitions from the components resp. classes to partition. A tool called Resharper was used to *automatically* extract all service interfaces from classes into new interface files.

2. Replacing direct method calls to the identified components respectively classes by proxy calls to a generated service proxy. For example, 7 service clients using `XERP.Server.DAL.MenuSecurityDAL` have been redirected via a proxy by a manual class instantiation with a call to the redirected class. The apparently redundant amount of program code could have been diminished by only instantiating a single class and using class methods instead of object methods. But the decision lead to a cleaner software code.

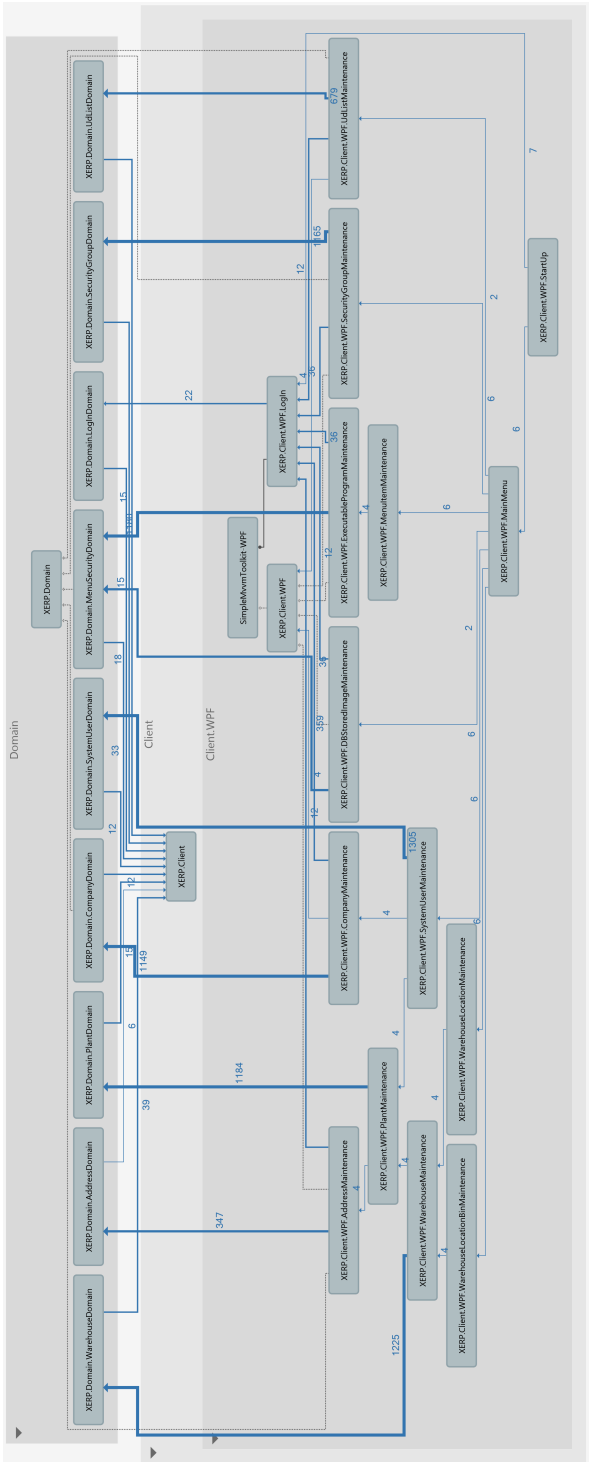


Figure 8.5: Visualization of XERP.NET's client's subsystem dependencies

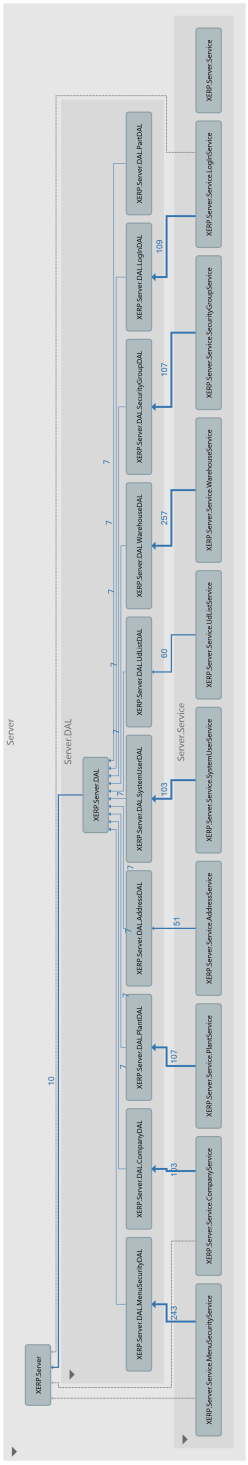


Figure 8.6: Visualization of XERP.NET's server's subsystem dependencies

3. Adapting the references to assemblies for the partition points according to a distributed architecture. Assemblies are artifacts of Microsoft .NET's module system.

4. Compiling the source code of XERP.NET and deploying the resulting program to an *internet information server* (IIS).

5. Compiling the generated service host source code including all partitioned parts of XERP.NET. This part did not need any modifications.

6. Adaption of the WCF configuration to an existing network topology (IP-addresses and ports).

7. Starting an IIS instance with XERP.NET and the generated service host.

8. Testing the functionality of XERP.NET by user interface tests and more thoroughly by performing provided unit tests.

8.3.3 Results

From both synthesized connectors we generated 1731 LLOC for WCF-configurations and C#-code in 18.11s. Minor (20 LLOC) manual adaptations in the original implementation of XERP.NET were necessary to use the generated proxies for redirection instead of the local components. Major adaptations to XERP.NET had to be made because XERP.NET was developed for earlier .NET, IIS, and SQL Server versions.

We deployed the distributed implementation of XERP.NET and executed and tested it on separate virtual machines. The resulting system showed no functional deficiencies after performing unit tests. The distribution of the system had an impact on non-functional properties, for example, a measurable performance reduction caused by higher network latency. This, however, is to be expected from a partition of the system. By making the earlier design decision to distribute XERP.NET we accepted this inevitable trade-off.

For an analysis and a discussion on this experiment, we refer to the end of this chapter for a detailed analysis and discussion of the experiments.

8.4 e-Commerce Scenario

The question for this experiment is, how effective and with how many manual adaptions can the Combinatory Logic Connector Synthesis method be applied to a realistic, large-scale software application that is not entirely streamlined

for our method. Furthermore, we want to analyze the scalability of our method on the software system's size and/or complexity.

8.4.1 Setup

We apply the methodology to a second real-world software system to evaluate its applicability and scalability. We chose an open-source e-Commerce solution, AspxCommerce V2.0[11] (AspxCommerce) as the respective test object. AspxCommerce as an application is realistic, because it is a commercially operated web shop solution by many small- and big-sized e-Commerce companies.

AspxCommerce consists of 37 subsystems with 1293 classes with 225 763 LLOC. The figures also include some subprojects contained in AspxCommerce. Standard components of .NET are explicitly excluded from the presented figures. Again, the goal is to distribute the monolithic object-oriented architecture/implementation of the software system and to synthesize connectors necessary to distribute the components into a network-based architecture. We defined the partition points for the system on the premise to minimize impacts on functional and non-functional system-properties. The architecture of AspxCommerce is such that all interactions between components are one-to-one. Thus, for the chosen partition points we only have to synthesize I-connectors replacing the method calls between the involved components.

8.4.2 Execution

We reuse the taxonomy and templates developed for the previous experiment on XERP.NET. Thus, the repository Γ used in the following experiment on AspxCommerce contains 24 building blocks out of the variety of building blocks mentioned above.

Figure 8.7 contains a visualization of the subsystem dependencies of AspxCommerce's created with an architecture tool in JetBrains Resharper 8.1, a refactoring tool for Visual Studio 2013. The visualized subsystems contain in median more than 6000 LLOC and nearly 35 classes per subsystem. The dependency view depicted here was used to identify partition points that are congenial to the condition of minimal dependency.

Again, we consider one partition point in more detail. AspxCommerce uses a module, `CurrencyConverter`, for currency conversion. Its interface *ICurrencyConverter* is used by various other modules, for instance, by the component `AspxCommerceWebService`, which exposes the functionality of AspxCommerce via a SOAP-web service. We separate `CurrencyConverter`

[11] `http://aspxcommerce.codeplex.com/`

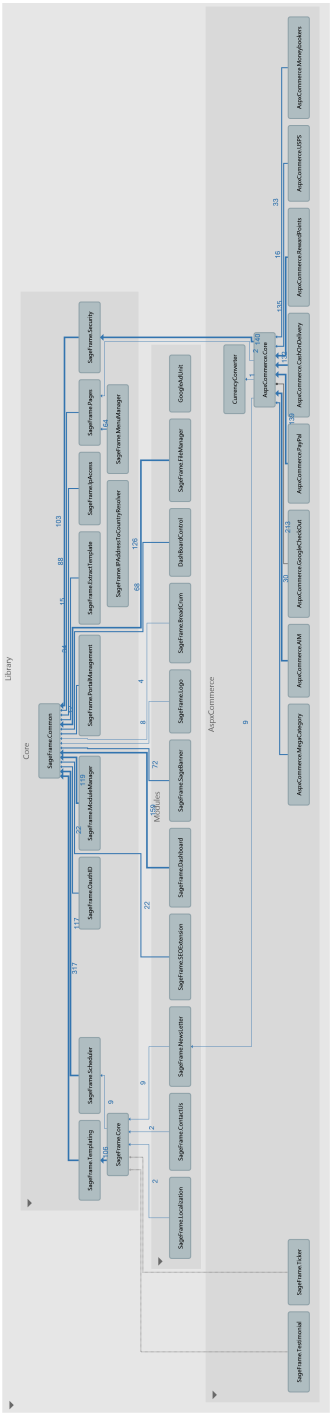


Figure 8.7: Visualization of AspxCommerce's subsystem dependencies

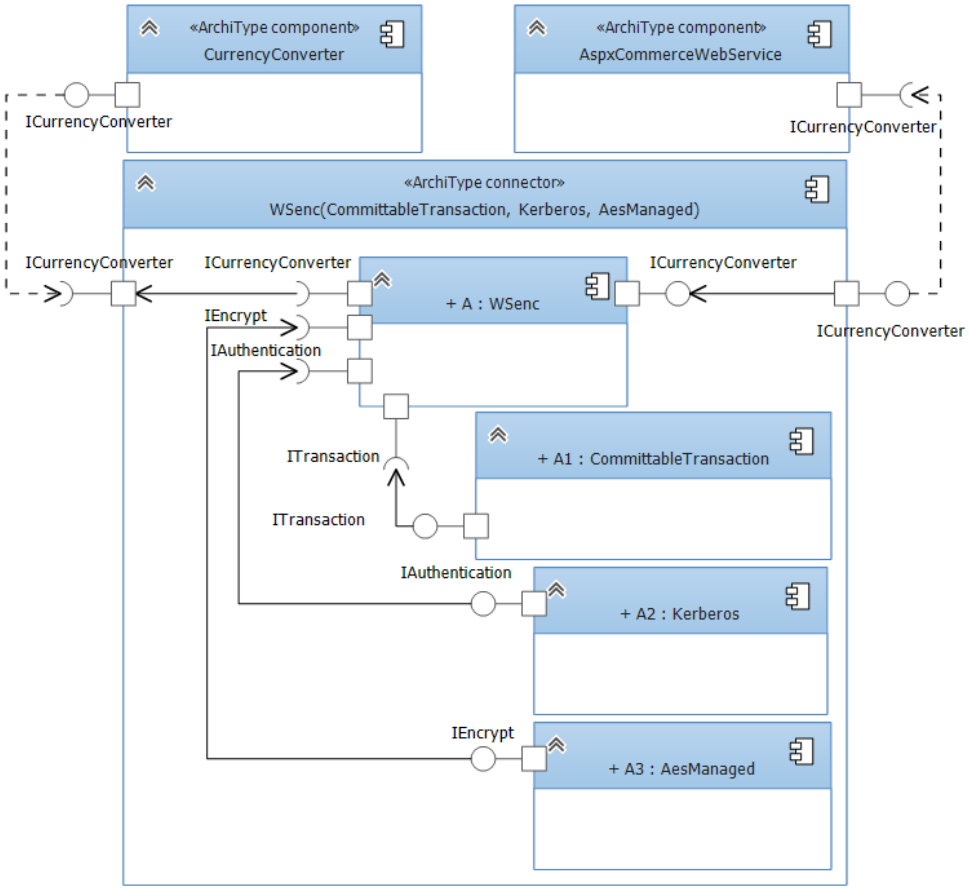


Figure 8.8: Generated UML2-component diagram in VS2013 for component CurrencyConverter

from the monolithic implementation of AspxCommerce and we require that the *ICurrencyConverter*-connector should be a SOAP-web service supporting transactions, authentication by Kerberos, and symmetric encryption.

Thus, we pose the corresponding inhabitation question

$$\Gamma \vdash ? : (\mathit{ICurrencyConverter} \to \mathit{ICurrencyConverter}) \\ \cap \mathit{ws} \cap \mathit{transactionEnabled} \cap \mathit{Kerberos} \cap \mathit{sym}.$$

The resulting inhabitant was used to automatically generate the UML2-component diagram shown in Figure 8.8.[12] The residual partition points were treated similarly (requiring other specific properties as needed) which resulted

[12]The ≪ArchiType connector≫-component wraps the generated connector and delegates the interfaces of ws_{enc}.

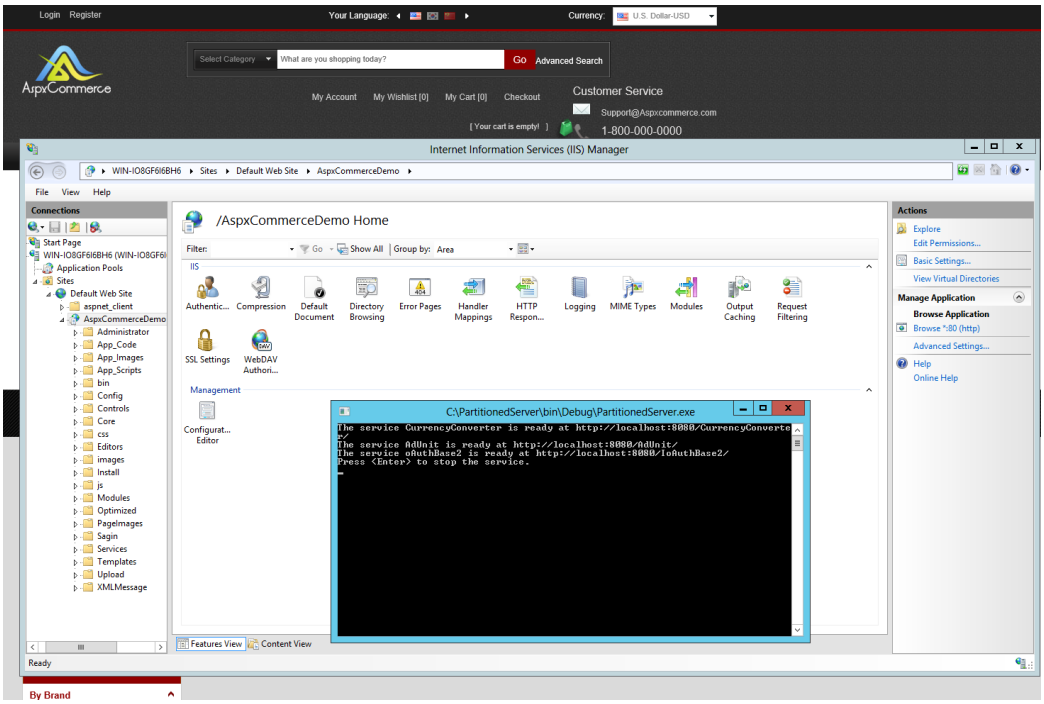


Figure 8.9: Screenshot of the configuration interface of the *internet information server* (IIS) hosting AspxCommerce visible in the background

in similar UML2-component diagrams. ArchiType used the enhanced diagram to generate the synthesized connectors using the combinators underlying templates.

The resulting source code was integrated and compiled. The compiled application was then deployed to a Microsoft Windows 2012 Server with an *internet information server* (IIS) and a Microsoft SQL Server 2012 as database back-end for AspxCommerce. The IIS hosted the AspxCommerce application. A screenshot in Figure 8.9 depicts the configuration interface of the IIS containing the AspxCommerce as default website. In the background AspxCommerce is visible in the browser window. The server is visible in the console output offering the three generated services with individual service-endpoints.

A generated server hosting the mentioned services was also deployed and executed. The screenshot in Figure 8.10 shows the running AspxCommerce application in a web-browser client (Microsoft Internet Explorer) with the generated server exposing all three services. The currency selector is visible in the top row as most right hand side graphical element. There, a customer can select the monetary currency used by AspxCommerce. The currency

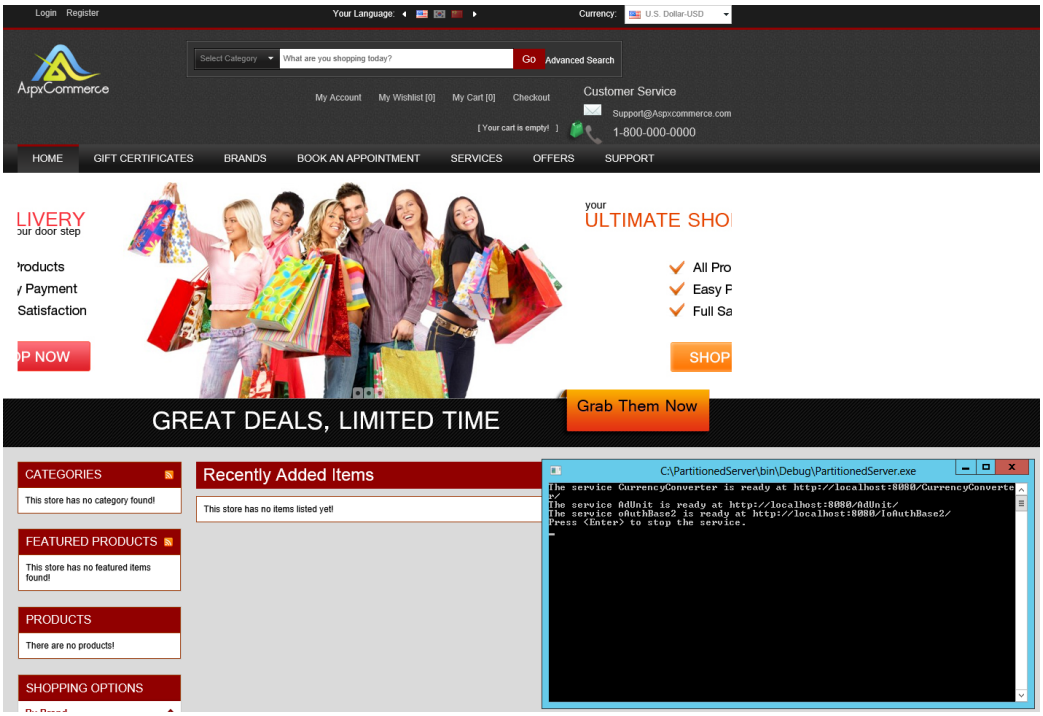


Figure 8.10: Screenshot of AspxCommerce including generated code

conversion is now redirected and performed by the generated server providing the `CurrencyConverter` service.

The following modifications to the original source code of AspxCommerce and manual operations had to be made:

1. Extracting service interface definitions from the components respectively classes to partition. A tool called Resharper was used to *automatically* extract all service interfaces from the classes into new interface files.

2. Replacing direct method calls to the components respectively classes by proxy calls to a generated service proxy.

3. Adapting the references to assemblies for the chosen partition points to reflect the intended distributed architecture.

4. Compiling the source code of AspxCommerce and deploying the resulting program to an IIS.

5. Compiling the generated service host source code including the partitioned parts of AspxCommerce. This part did not need any modifications.

6. Adaption of the WCF configuration to an existing network topology (IP-addresses and ports).

7. Starting an IIS instance with AspxCommerce and the generated service host.

8. Testing of the functionality of AspxCommerce by user interface tests and more thoroughly by performing provided unit tests.

Some typical source code modifications are depicted for the component `CurrencyConverter`. We omit implementation details and present some key modifications that have been conducted. Before interface extraction, the declaration part of `CurrencyConverter` was:

```
1  public class CurrencyConverter
2  {
3    public double ConvertCurrency(string from, string to, double
         amount)
4    {
5      // ...
6    }
7
8    public double GetRate(string from, string to)
9    {
10     // ...
11   }
12
13   public double GetRateFromGoogle(string fromCurrency, string
         toCurrency)
14   {
15     // do a web service call to google's service...
16   }
17
18   public double GetRateFromYahoo(string from, string to)
19   {
20     // do a web service call to yahoo's service...
21   }
22 }
```

After refactoring all classes with an automatic interface extraction tool, an additional interface definition for `ICurrencyConverter` in Lines 1 to 12 has been generated and `ICurrencyConverter` implements this interface in Line 14. Then, the corresponding source code is modified:

```
1  [ServiceContract]
2  public interface ICurrencyConverter
3  {
4    [OperationContract]
5    double ConvertCurrency(string from, string to, double amount);
6    [OperationContract]
7    double GetRate(string from, string to);
8    [OperationContract]
9    double GetRateFromGoogle(string fromCurrency, string toCurrency);
10   [OperationContract]
11   double GetRateFromYahoo(string from, string to);
12 }
13
14 public class CurrencyConverter : ICurrencyConverter
15 {
16   public double ConvertCurrency(string from, string to, double
         amount)
17   {
18     // ...
19   }
20
21   public double GetRate(string from, string to)
22   {
23     // ...
24   }
25
26   public double GetRateFromGoogle(string fromCurrency, string
         toCurrency)
27   {
28     // do a web service call to google's service...
29   }
30
31   public double GetRateFromYahoo(string from, string to)
32   {
33     // do a web service call to yahoo's service...
34   }
35 }
```

The source code for a method call to `CurrencyConverter` is shown in the following listing in class `AspxCommereceWebService` lines 7341 to 7378.

```
1  [WebMethod(EnableSession = true)]
```

```
2  public double GetCurrencyRateOnChange(AspxCommonInfo
       aspxCommonObj, string from, string to, string region)
3  {
4    System.Net.ServicePointManager.Expect100Continue = false;
5    try
6    {
7      // ...
8      if (isRealTimeEnabled.ToLower() == "true")
9      {
10       try
11       {
12 //      result = (new
         AspxCommerce.Core.CurrencyConverter()).GetRate(from, to);
13         result = (new
               CurrencyConverterClient("DefaultBinding_ICurrencyConverter",
               "http://localhost:8080/CurrencyConverter/")).GetRate(from,
               to);
14       }
15       catch (Exception)
16       {
17         return 1;
18       }
19     }
20     else
21     {
22       result =
             AspxCurrencyProvider.GetRatefromTable(aspxCommonObj, to);
23     }
24     HttpContext.Current.Session["CurrencyCode"] = to;
25     HttpContext.Current.Session["CurrencyRate"] = result;
26     HttpContext.Current.Session["Region"] = region;
27     return Math.Round(double.Parse(result.ToString()), 4);
28   }
29   catch (Exception ex)
30   {
31     throw ex;
32   }
33 }
```

Line 12 contains the original method call to the `CurrencyConverter` commented (out) and Line 13 contains the redirected call to the generated proxy

for `CurrencyConverter` that is using the new service hosted by a server application.

8.4.3 Results

We examine some figures concerning the synthesis of these software connectors. If we pose an inhabitation question for synthesizing a most general I-connector, i.e., we ask the inhabitation question $I \to I$ without specifying any restricting properties, then from Γ we obtain 1920 different inhabitants. Of course, this does not present a practical solution, as must be expected when asking for a completely general (underspecified) software connector. So we went on to specify the connectors to narrow down the set of solutions. In our experiments we were able to refine the specifications of the repository and the desired connectors to receive unique solutions, for example, the inhabitation question stated above for the synthesis of a connector between the `CurrencyConverter` and `AspxCommerceWebService` resulted in a single inhabitant.

From the three synthesized connectors we generated 2171 LLOC for WCF-configurations and C#-code in 21.98s. Minor (5 LLOC with ratio 0.000 02) manual adaptations in the implementation of AspxCommerce were necessary to use the redirection provided by the generated proxies instead of the local components. Again, major adaptations to AspxCommerce had to be made because AspxCommerce was developed for earlier .NET, IIS, and SQL Server versions. We deployed the distributed implementation of AspxCommerce and executed and tested it on separate virtual machines. The resulting system showed no functional deficiencies under performing various unit tests.

8.4.4 Analysis and Discussion

By means of the resulting distributed implementation of AspxCommerce we illustrate one additional feature of (CL)S. We generated three separate servers, one for each partition point. Depending on the requirements, it might have been sufficient to only generate a single server that exposes the interfaces of all three components as services. It is not difficult to incorporate this adapted implementation into our approach. The type-based description of an interface *only* describes the intended usage of the involved components, but it leaves out any details as to *how* the functionality is implemented by underlying templates. Thus, if a different implementation of the specified functionality (for instance, one server as opposed to three servers) is required, we could simply replace the underlying templates without any need to change the type-based specification of the combinators. It allows for great flexibility,

since we may use sockets instead of WCF for messaging. Similarly, we may change other implementation details or even the used programming language.

8.5 Collective Analysis and Discussion

We summarize some facts that can be concluded from the experimental results in this chapter as follows. Table 8.1 depicts some results of the conducted experiments described in this chapter.

	eCommerce	ERP	Broker	Secure Connector
Subsystems	37	137	11	3
Classes	1293	691	11	3
LLOC	225 763	38 250	839	37
Generated/LLOC	2171	1731	993	257
Time/s	21.98	18.11	7	8.1
Refactoring/LLOC	5	20	0	0

Table 8.1: Experimental results for various applications of ArchiType

8.5.1 Expressiveness

Clearly, the design of suitable C&C environments using $\mathsf{BCL}_0(\cap, \leq)$ needs getting used to. But some basic experience in logic programming facilitates such a design. It should also be noted that the succinct encoding and the mathematical representation makes the design easy. The relational data-model of ArchiType that is stored in the SQL Server database is close (isomorphic) to the set-theoretic model presented in Section 6.2 on page 120. ArchiType uses this object model to automatically map it to a corresponding C&C type environment.

We would like to note that, in particular, the intersection of different specifications is very powerful for expressing a finite set of variants a building block can have. For example, if we go back to the detailed experiment in which a broker for providing a brokerage service to various clients and servers was synthesized, then we have the problem that the exact number of clients and servers varies in different scenarios. The exact number of clients is needed for the representation of the container building block as an intersection type. Clearly, a first approach would be to define the container building block in a way that it exactly fits to the use in a given scenario. This would have the

disadvantage that this supplementary definition of a building block has to be repeated for every scenario and this would also possibly lead to a definition of various sets of templates. By using the intersection, a container building block can be specified with a combination of finitely many specifications of its intended future usage. It is easy to see that corresponding templates can be specified polymorphically to allow variable sets of client and servers, for example, by assigning a list of clients instead of a single client.

Nevertheless, a repository has to be designed for composition requiring a non-trivial degree of design intelligence, abstraction, and specification. Smartness is more located in the design and specification of the repository and to a lesser extent in the (software-)components.

8.5.2 Applicability

Here it should pointed out that in all experiments described before, only minor manual adaptations and *no* redesign of the applications was necessary. In XERP.NET only 20 LLOC and in AspxCommerce only 5 LLOC had to be adapted. Both applications are not ideal for our Combinatory Logic Connector Synthesis method presented here, because both applications are POCO applications containing components without explicitly exposed interfaces. Both applications have been chosen, in particular, because we were able to apply our methodology even to such applications in POCO style easily. Even in this case, only minor adaptions to components or configurations had to be made.

8.5.3 Adaptability

ArchiType has been applied to some more (minor) scenarios. For one of these scenarios, another communication framework with a different technology has been integrated. ServiceStack[13] differs from WCF in many points but especially in the style of extensibility. We suspected that some bigger changes or modifications had to be made. However, only minor modifications had to be made, because both communication framework share a compositional approach for extensibility.

The adaptability of the Combinatory Logic Connector Synthesis method strongly relies on the adaptability of implementation templates that are linked to building blocks. As described in the subsection "limitations", different design principles, e.g. open/closed-principle by Meyer [1988], allow for designing more adaptive components. Various approaches has also been discussed

[13]ServiceStack web page: `https://servicestack.net/`

and analyzed for the related work on component-based software engineering by Heineman [2000], on adaptable software components by Heineman and Councill [2001], and on generative software development by Czarnecki [2004] as well as Czarnecki and Eisenecker [2000]. Such related results are directly transferable to our method because our method do not pose any further restrictions to target programming languages for generation.

The line of work around meta-programming, e.g. in particular for staged meta-programming like for MetaML [Taha and Sheard, 2000, Pitts and Sheard, 2004] or more recently with hygienic code generation for Haskell [Kameyama et al., 2014], provides different principles and ideas for adaptable components and program code generating program code.

Here, we want to spotlight that the synthesis and generation steps of our method are agnostic to different ADLs and also to varying programming languages, for instance Java, and related technologies as long as these support composition.

8.5.4 Costs and Effort

One could argue that it is not clear that this methodology poses an advantage to manually implementing the needed software connectors. We can discriminate between *design costs*, (manual) *adaptation costs*, and *direct implementation costs*. The methodology's *efficiency* can be measured by the ratio of the direct implementation costs to the sum of design and adaption costs.

The last three experiments shared the same and unchanged C&C type repository as well as underlying code templates. The design cost for these experiments is the sum of code template sizes (791 LLOC). The adaption costs have been 25 LLOC. But, reusing these code templates in the three last experiments, 4895 LLOC have been generated automatically. For every logical code line in the code template, approximately 6 logical code lines in the connector have been generated automatically.

And indeed, we conducted many more experiments on software connectors with different inhabitants.[14] These auxiliary experiments produced no further costs, because the before described code adaptions had already been made and was reused.

The evolution of a generated software architecture can be traced by using a feedback loop. The feedback loop augments an existing C&C type environment by synthesized inhabitants with possibly supplementary speci-

[14]Currently more than 200 times software connectors have been generated successfully by ArchiType in various experiments.

fications. An already synthesized inhabitant in the C&C type environment would directly reflect the current state of the software architecture. Newly posed inhabitation questions, for example a connector synthesis goal, could use already synthesized inhabitants to synthesize an incremental transformation specification Δ which applied to a current software architecture state transforms it into a goal state specified by the posed inhabitation questions. Complementary, such a feedback loop could be formed into a Deming cycle (or *plan–do–check–act* (PDCA) cylce), for example discussed in [Deming, 2000, page 88]. The Deming cylce is a well-established management method consisting of four steps. Each step can be assigned a suitable semantic in our synthesis approach.

1. The plan step of the Deming cycle defines a new inhabitation question.
2. The do step performs the synthesis and generation of an incremental transformation specification Δ.
3. The check step performs an evaluation of the generated system including the transformation specified by Δ.
4. And finally, in the act step, the evaluation is analyzed, a new system goal is defined and measures are derived to achieve the new system goal.

Applying the Deming cycle helps to control and continuously improve a software architecture.

8.5.5 Usability

Our tool ArchiType is far from being usable in daily industrial software engineering. Some improvements could be considered for future releases.

The user interface bears some improvements. One improvement would be an easier comprehensible presentation of the various inhabitants that could be synthesized as software connectors. The applicative term representation could be replaced by a more intuitive graphical representation showing a preview of the currently selected inhabitant, for example as a UML2 component diagram of a selected inhabitant.

An analogous representation idea can also be applied to the design and management of a C&C type environment that is stored in a SQL Server database. A management GUI would simplify the maintenance and extension of C&C type environments.

The code generation process could be integrated into the model-transformation-pipeline of Visual Studio 2013. Such a higher degree of integration

would enable the usage of additional provided tools and frameworks in Visual Studio 2013.

8.5.6 Limitations

The Combinatory Logic Connector Synthesis method, presented here, clearly has limitations.

The design of the C&C type environment is entangled with the development of corresponding code templates. Such an entanglement requires a strict discipline of a designer as well as a developer. In particular, obeying the following four object-oriented programming principles aid to limit the scope of needed design decisions.

- The single responsibility principle by Martin [2002] states that every component should have a single responsibility which is entirely encapsulated by the component. It means transfered to our method that each building block and an associated template (from a linked of set of templates) is only liable for a single functionality. In particular, this is necessary, because it would not be clear, how to specify meaningful properties of such building blocks using intersection types.

- The open/closed principle introduced by Meyer [1988] states that "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". This principle is important especially for the composition of our building blocks and templates. In particular, the development of the templates represents a challenge to reach a highest degree of extensibility of the template.

- The Liskov substitution principle, which is closely related to type theory, was introduced by Liskov and Wing [1994] stating "Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S where S is a subtype of T." The Liskov substitution principle is important for functional application just as well as for designing an appropriate C&C type environment and an associated set of templates. The principle is naturally included in our method by the usage of type theory. The arguments of an arrow $\rightarrow$ type are contravariant whereas the target of the type is covariant. Wing's behavioral types can be simulated in intersection types by introducing types representing behavioral concepts.

- The interface segregation principle by Martin [2002] states that no component should be forced to depend on methods it does not use.

> This principle suggests to refine the interfaces by splitting the interfaces of the component which are very large into smaller and more specific ones. Then, these more specific interfaces will be used by clients that are of interest for them. Such specific interfaces can be refined even more and therefore made more specific by attaching semantic concepts using intersection types.

The remaining SOLID principle, the dependency inversion principle, does not apply to our method. Additionally, the combinators and templates have to be developed under a minimal assumption on the later usage of these elements but with a maximum size of possible compositionality.

Our method is developed with the intention of adding value by reusing components from a repository for many projects or scenarios. Therefrom, clearly our method does not bring a value added applied to one-time projects, because the costs described before do not amortize the generated value by using our method.

Undoubtedly, projects or scenarios containing heterogeneous programming languages, technologies, or frameworks as variability dimensions constitute a further limitation, because such dimensions are not encompassed by the method and need different subsets of templates to cover all these dimensions.

A self-imposed restriction to our method is the synthesis and generation of behavior. In Chapter 2 about software architecture we discussed the differences between components and connectors. A notable difference is the difference of computation vs. interaction. Our method can be extended for supporting the synthesis and generation of behavioral connectors and also of selected components. This extension is limited by the requirement that we must hide a behavior in an abstract specification.

A cornerstone of this thesis are repositories containing specified or typed components. Such a repository of typed components can be perceived as an API. Using APIs as programming mechanism for distributed systems provide location transparency by facilitating local method calls that are hiding network operations. Such APIs as interfaces for distributed systems also necessitate an intense coupling between actors. It only admits two-party client/server architecture and implies a rigid knowledge of the exchanged data structure. A usage of asynchronous communication exchange patterns are often discouraged even though needed for advanced distributed systems. APIs, as a class of technology, disclaim the fundamental complexities of distributed systems like

- coping with network failures,
- making relevant consistency choices, and

- obfuscate the determination of influences of data representations on various properties of distributed systems like availability.

All technologies presented here, e.g. WCF, represent a hybrid solution because these technologies provide an API and therefore provide a transparent view on distributed communication but also provide configurations and restricted behavioral adaptations allowing for a fine tuning of distributed algorithms. A more fine grained support of distributed algorithms necessitates as well a more detailed knowledge about computations within a component influencing properties of a distributed system like logical non-monotonic operations. Therefore, the direct approach presented here is not appropriate for general distributed systems.

Chapter 9

Conclusion

We conclude this thesis with a résumé comprising a retrospective and an ex post facto on the results presented before and a prospective on future work.

Résumé

The thesis presents a step towards closing the gap to practical applicability of (CL)S by presenting heuristical optimizations of the $\mathsf{BCL}_k(\cap, \leq)$-inhabitation algorithm and by applying the framework to software synthesis on a large scale for the first time. It should be understood that the Combinatory Logic Connector Synthesis method does not present an approach from which readily deployable software is to be expected. Rather we view the approach (and (CL)S in particular) as a rapid prototype-generation framework, ArchiType. For a given scenario prototypical implementations of software connectors, that may possibly require some manual adaptation, can be generated rather quickly. From such prototypes it is often possible to estimate feasibility of ideas for software-solutions much more precisely before implementing them in detail.

It is clear that in designing the repository the semantic specification of the building blocks by intersection types and the definition of suitable underlying templates needs sophistication and intelligence. In some cases our experiments revealed interesting solutions that were not expected, in other cases specifications needed to be specified more detailed to narrow down solutions. We verified that the tool and experiments may be used to refine the specification of a repository. Altogether, the increased effort necessary for specifying the repository pays off if the repository is reused for repeated synthesis requests. Thus, (CL)S is particularly suitable for synthesis tasks that rely on repeated uses of the same repository.

Future Work

In future work we want to push our research further in both directions that have been pursued in this thesis. The reduction of the number of substitutions that have to be instantiated during the execution of the type inhabitation algorithm is directly related to its execution time. We plan to apply further heuristic filtering of substitutions early on.

The current implementation of the inhabitation algorithm in (CL)S uses a strict linearizability for consistency. This consistency requirement comes with a cost of an immense communication overhead in a distributed environment. By relieving this requirement to eventually consistency, a distributed inhabitation algorithm could exploit the consistency and logical monotonicity (CALM) theorem by Conway et al. [2012] for better performance. In addition, *convergent or commutative replicated data types* (CRDTs) can be applied to various data structures in a distributed implementation and further reduce needed communication effort.

We want to further substantiate the applicability of (CL)S by performing further experiments. We will do so by continuing work on connector synthesis (other ADLs, generation of system management code for software deployment, representation of protocols by behavioral types [Gay, 2012] (ICT COST Action IC1201 – Behavioural Types for Reliable Large-Scale Software Systems (BETTY)), etc.) but we will also look for other scenarios that are amenable for component-based (software) synthesis (for instance, code synthesis for embedded systems, business processes, and workflows).

Another interesting application field for synthesis are dynamic typed languages that are ubiquitously used in web-based environments and scenarios. The proof theoretic work by Henglein [1994] on dynamic typing and liquid types by Rondon et al. [2008] could be starting points for an extension of our type inhabitation algorithm and (CL)S for dynamic languages.

In an additional line of future work, we want to synthesize state-dependent components like mixins or objects. Mixins and objects might add a novel stage, runtime or execution, to the already existent composition and compilation stage presented in the staged computations synthesis approach. Furthermore, it is not clear how a type-theoretic model of such a stage looks like.

It will also be interesting to see whether we can integrate our approach with other code generation frameworks and to investigate how higher-order combinators can be used to describe how other code templates are to be combined. A further development could be the integration of behavior in this method. Intersection types and type inhabitation can be used to simulate finite state machines that build the fundamental model in which behavior can be specified.

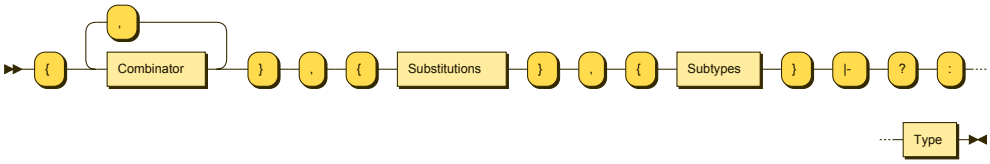
Appendix A

Selected implementation details

A.1 Grammar of (CL)S's input language

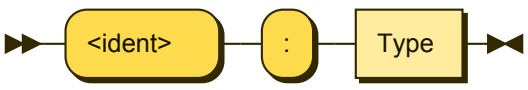
The grammar in EBNF as railroad diagram and in textual representation of EBNF is defined as follows:

Grammar



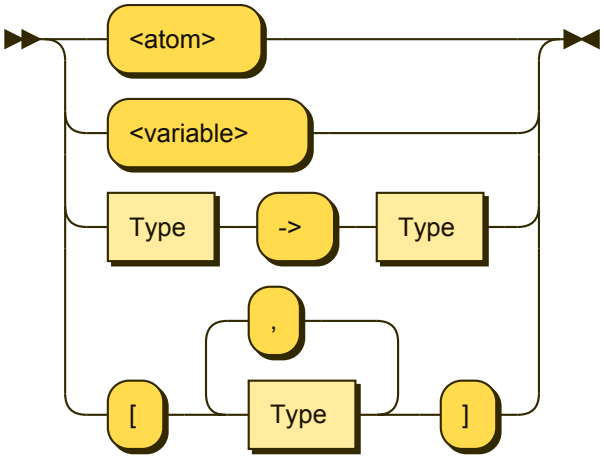

```
Grammar  = '{' Combinator ( ', ' Combinator )* '}' ','
           '{' Substitutions '}' ','
           '{' Subtypes '}' '|-' '?' ':' Type
```

Combinator



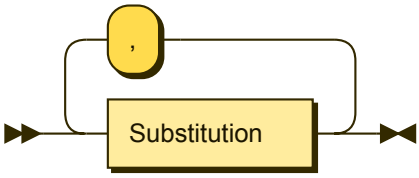

```
Combinator = '<ident>' ':' Type
```

Type



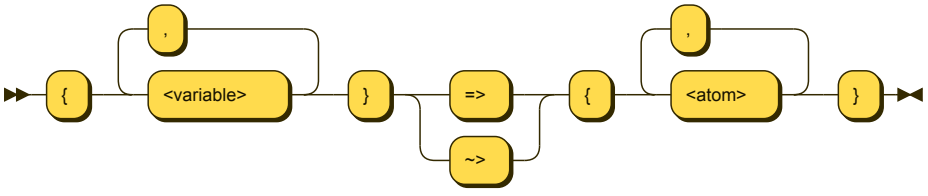

```
Type      = '<atom>'
          | '<variable>'
          | Type '->' Type
          | '[' Type ( ',' Type )* ']'
```

Substitutions



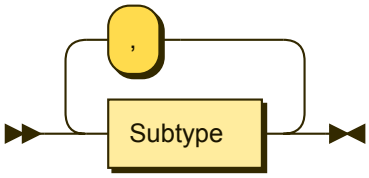

```
Substitutions = Substitution ( ',' Substitution )*
```

Substitution




```
Substitution = '{' '<variable>' ( ',' '<variable>' )* '}'
               ( '=>' | '~>' )
               '{' '<atom>' ( ',' '<atom>' )* '}'
```

Subtypes




```
Subtypes = Subtype ( ',' Subtype )*
```

Subtype


```
Subtype = '<atom>' '<=' '<atom>'
```

The lexer rules are omitted because these rules are directly derivable from the EBNF. The implementation of the lexer and parser has been done using the F# modules FsLex and FsYacc of the F# Powerpack.[1]

A.2 Example in the Listing in ArchiType CL

```
1  // ------------------------------
2  // Declaration of templates with its formal parameters
3  // ------------------------------
4  GenerateProxyStub(libraryDLL,workingDir)
5  GenerateServer(componentName, serverSettings)
6  GenerateServerConfig(serviceName, serviceInterface, behaviorTemplate,
       bindingTemplate, bindingName)
7  GenerateClientConfig(serviceName, serviceInterface, behaviorTemplate,
       bindingTemplate, bindingName)
8  GenerateSecurity(componentName)
9  GenerateClientSettings(namespace, className, connectorInitializer)
10 GenerateClientProxyCode(referenceToInterface, proxyClass)
11 GenerateBinding(bindingName, bindingEntries, bindingProperties)
12 SharedSecret()
13 EnableTransaction()
14 EnableTransactionClient()
15 #
```

[1] Projects main webpage: `http://fsharppowerpack.codeplex.com/`.

```
16 // ------------------------------
17 // Declaration of variables with their initial values
18 // (the values are set by ArchiType)
19 // ------------------------------
20 $interfaceFile=".\SimpleExampleInterfaceB.dll"
21 $namespace="SimpleExample"
22 $classServer="ClassB"
23 $classClient="ClassA"
24 $interfaceName="InterfaceB"
25 $bindingName="basicHttpBinding" // choose basicHttpBinding
26 $libraryPath= ".\AspxCommerceV2.0_Source\SageFrame\bin\"
27 #
28 // ------------------------------
29 // Code generation code is generated by an reduction of an inhabitant
30 // ------------------------------
31 // class and interface fully qualified
32 $qualifiedClassServer=$namespace+"."+$classServer
33 $qualifiedInterface=$namespace+"."+$interfaceName
34
35 //Inhabitant: WSenc(SharedSecret, EnableTransaction)
36 // generate proxies & stubs
37 GenerateProxyStub($libraryPath+"CurrencyConverter.dll",
       ".\GeneratedFiles\CurrencyConverter")
38   ->>".\GeneratedFiles\CurrencyConverter\Test.txt"
39 GenerateProxyStub($libraryPath+"SageFrame.OauthID.dll",
       ".\GeneratedFiles\OauthID")
40   ->>".\GeneratedFiles\OauthID\Test.txt"
41 GenerateProxyStub($libraryPath+"SFE.GoogleAdUnit.dll",
       ".\GeneratedFiles\GoogleAdUnit")
42   ->>".\GeneratedFiles\GoogleAdUnit\Test.txt"
43 // generate server code in C#
44 GenerateServer("CurrencyConverter$AdUnit$oAuthBase2", "//TODO:
       Change connection settings")
45   ->>".\GeneratedFiles\Server\Server.cs"
```

A.3 Example Template in T4

The listing produces a C# server that provides different services whose names are provided via the parameter `componentName`. Each of these names is separated by `$`.

```
1 <#@ template language="C#" debug="true"#>
```

```
2  <# //Uncomment this line to test that the host allows the engine to set
       the extension. #>
3  <# //Uncomment this line if you want to debug the generated
       transformation class. #>
4  <# //System.Diagnostics.Debugger.Break(); #>
5  <#@ parameter name="componentName" type="System.String" #>
6  <#@ parameter name="serverSettings" type="System.String" #>
7  using System;
8  using System.ServiceModel;
9
10 namespace SimpleExample
11 {
12     class Program
13     {
14     static void Main()
15         {
16         // Create the ServiceHost.
17         <#
18         var services = componentName.Split('$');
19         foreach (var service in services)
20             WriteLine(String.Format("using (var host{0} = new
                   ServiceHost(typeof({0})))\n{{", service));
21         #>
22         try
23         {
24             // Server settings section
25             <#=serverSettings#>
26             // Open the ServiceHost to start listening for messages. Since
27             // no endpoints are explicitly configured, the runtime will create
28             // one endpoint per base address for each service contract
29             // implemented by the service.
30             <#
31             foreach (var service in services)
32             {
33                 WriteLine(String.Format("host{0}.Open();", service));
34                 WriteLine(String.Format("foreach(var uri in
                       host{0}.BaseAddresses)", service));
35                 WriteLine(String.Format("\tConsole.WriteLine(\"The service
                       {0} is ready at {{0}}\", uri);", service));
36                 WriteLine("// Install a failure handler for logging the error");
```

```
37              WriteLine(String.Format("\thost{0}.Faulted+=HostFaulted;",
                    service));
38          }
39          #>
40          Console.WriteLine("Press <Enter> to stop the service.");
41          Console.ReadLine();
42          // Close the ServiceHost.
43          <#
44          foreach (var service in services)
45              WriteLine(String.Format("host{0}.Close();", service));
46          #>
47      }
48      catch (TimeoutException timeProblem)
49      {
50          Console.WriteLine(timeProblem.Message);
51          Console.ReadLine();
52      }
53      catch (CommunicationException commProblem)
54      {
55          Console.WriteLine(commProblem.Message);
56          Console.ReadLine();
57      }
58      <#
59      foreach (var service in services)
60          WriteLine(string.Format("}} // closing using service {0}",
                service));
61      #>
62  }
63
64  private static void HostFaulted(object sender, EventArgs e)
65  {
66      Console.WriteLine(e.ToString());
67  }
68 }
```

A.4 Algorithm Listings

A.4.1 Inhabitation Algorithm (data-centric part)

The algorithm is implemented in Microsoft F#. The implementation is very close to the optimized algorithm presented as pseudo-code (of an ATM) in Figure 4.7 on page 96.

Listing A.1: F# function inhabBCLkOptimized in (CL)S

```
1  let inhabBCLkOptimized env typeConsts substs atomicSubtypes tau
       (node : Node) (adder : System.Action<QueueEntry>)=
2      let tau2=Organize tau
3      let typeVars= projectTypeVarsOutOfSubstitutions substs
4      let PossibleSuccessList= env |> List.map (fun x->
5          let usedTypevars= extractUsedTypevars (snd x) typeVars
6          let usedSubstitutions= filterUsedSubstitutions substs usedTypevars
7          let NAndTauCSigmaCandidates= matchingSigmasWithTaus
               typeConsts usedTypevars usedSubstitutions (snd x) tau2
               atomicSubtypes
8          let candidates= snd NAndTauCSigmaCandidates
9          let N = fst NAndTauCSigmaCandidates
10         if not (List.isEmpty candidates) then
11             List.map (fun n->
12                 let TgtsAndArgs=generateTgtsAndArgsWithTau
                       candidates usedTypevars usedSubstitutions n env
                       typeConsts atomicSubtypes typeVars substs
13                 if List.forall (fun comp->not (List.isEmpty (snd comp)))
                       TgtsAndArgs then // Line 22 in Algo 5
14                     if n=0 then
15                         let newGroup= node.AddNewChildrenGroup(fst
                               x, 0, StatusType.Success)
16                         do node.Check(newGroup.GroupID);
17                         true
18                     else
19                         let sigmas= List.map (fun x->snd x)
                               TgtsAndArgs
20                         let crossProduct= cartesianProduct sigmas
21                         crossProduct |> Seq.iter (fun argFromCross->
22                             let args= Seq.map (fun i->
                                   (Intersect(List.map (fun p'->argPi p' i)
                                   argFromCross) |> Normalize)) (seq [1..n])
```

```
23                                  let newGroup=
                                        node.AddNewChildrenGroup(fst x, n,
                                        StatusType.Unknown)
24                                  args |> Seq.iter (fun arg->
25                                      let newNode= new
                                            Node(newGroup.GroupID,
                                            StatusType.Unknown, node,
                                            arg.GetHashCode(), node.FailCache,
                                            node.SuccessCache)
26                                      do newGroup.Children.Add(newNode);
27                                      let newEntry= new QueueEntry(node.ID,
                                            arg, (fst x), newNode)
28                                      do newNode.QueueEntry<-newEntry
29                                      do adder.Invoke(newEntry)
30                                      )
31                                  )
32                              true
33                      else
34                          false
35                          ) N |> Seq.exists (fun x->x=true)
36          else
37              false
38                                  )
39      let PossibleSuccess= Seq.exists (fun x->x=true) PossibleSuccessList
40      if not PossibleSuccess then
41          node.Status<-StatusType.Fail
42          node.CheckFail(node.GroupID)
43      PossibleSuccess
```

Appendix B

Experiments

This chapter contains more detailed information on the experiments that have been conducted.

B.1 Compiler

The following two compilers have been used to generate all executables programs for the experiments.

- **F#**: Microsoft (R) F# 2.0-Compiler, Build 4.0.40219.1
- **C#**: Microsoft (R) Visual C# Compiler Version 4.0.30319.18408 for Microsoft (R) .NET Framework 4.5

B.1.1 Compiler Flags

In the F# compiler the *optimize*, *tailcalls*, and *crossoptimize* flags were turned on and the *generate overflow checks* flag was turned off. Similarly, in the C# compiler the *optimize* flag was turned on and the *generate overflow checks* flag was turned off.

B.2 Benchmark Problems

The problems that have been used for the experiments on the parallelization strategies for RQS and TCP are: The example in Rehof [2013] is a scenario in which a tracking service has to be synthesized that tracks location and other information of transport containers as described in Section 1.3 on page 7. An appropriate composition has to be found. In this example, a composite function (as an applicative term) has to be synthesized that returns

Problem #	Problem
Problem 1	Example from Rehof [2013] and Section 1.3 on page 7
Problem 2	Γ_4^2
Problem 3	Γ_4^3
Problem 4	Γ_3^4

Table B.1: Problem definitions used for the experiments in the Subsection 4.2.4 on page 87

a Cartesian coordinate as real number ($\mathrm{R} \cap \mathrm{Cx}$). The repository is shown below:

```
{
Void : Bottom,
Tr : Bottom -> data -> ([ pair -> (R -> pair) -> R , Cart]
-> [R, Gpst] -> data) -> [R, Cel],
pos : (data -> ([pair -> (R -> pair) -> R,varCoord] ->
[R, varTime] -> data) -> R) -> [pair -> ([pair ->
(R -> pair) -> R, varCoord] -> pair) ->
[R, varTime], Pos],
cdn : [pair -> ([pair -> (R -> pair) -> R, varCoord] -> pair)
-> R, Pos] -> [pair -> (R -> pair) -> R, varCoord],
fst : [[pair -> (R -> pair) -> R , Coord ] -> R,
Cart -> Cx, Polar -> Radius],
snd : [[pair -> (R -> pair) -> R , Coord ] -> R,
Cart -> Cy, Polar -> Angle],
tmp : (data -> ((pair -> (R -> pair) -> R) -> R -> data) ->
[R , varTemp] ) -> [R , varTemp],
cc2pl : [pair -> (R -> pair) -> R , Cart] ->
[pair -> (R -> pair) -> R , Polar],
cl2fh : [R , Cel] -> [R , Fh]
},
{{varCoord} ~> {Coord, Cart, Polar}, {varTime}~>{Time, Utc, Gpst},
 {varTemp}~>{Temp, Fh, Cel}},
{ Cart <= Coord, Polar <= Coord, Gpst <= Time, Utc <= Time,
Cel <= Temp, Fh <= Temp }
|- ? : [R, Cx]
```

B.3 Configuration of the Test Systems

This section lists the information on the used test systems.

B.3.1 Test System I - Desktop PC

The experiments were performed on a single test system and not on a computing cluster to make measurements more controllable and reproducible. The test system has the following specifications:

- Microsoft Windows 8 Enterprise (Version 6.2.9200 Build 9200)
- Processor Intel(R) Core(TM) i5 CPU 750 2.67GHz 4 Cores
- Installed physical memory (RAM) 8.00 GB
- Microsoft Visual Studio Ultimate 2013
- Microsoft .NET Framework – Version 4.5.50743

B.3.2 Test System II - Compute Server

Some experiments were performed on a compute server cluster to make measurements more controllable and reproducible. The test system has the following specifications:

- Microsoft Windows Server 2012
- Processors 4 x AMD Opteron 6272 2.1GHz 12 Cores 16MB L3 Cache
- Installed physical memory (RAM) 96GB
- Microsoft Visual Studio 2013 Ultimate
- Microsoft .NET Framework – Version 4.5.50743

The compute server was virtualized with Linux Debian 6 as host operating system and Microsoft Windows Server 2012 as guest operating system. The virtualization infrastructure was Linux KVM with OpenNebula as Cloud Computing Stack. For better comparability of the results a specific instance with fixed resource guarantees for CPU and RAM was used. Despite these guarantees, the deviations were significantly greater than the deviations occurring in the isolated desktop PC in Section B.3.1. Furthermore, at least a 10% performance loss, caused by the virtualization with KVM, must be taken in consideration. Hence, the measurements have been made with relative figures, to eliminate such influences on the experimental results and to enhance the comparability of these results.

Acronyms

AADL architectural analysis and design language 21

ADL architecture description language 11

AI artificial intelligence 49

API application programming interface 7

ATM alternating Turing machine 43

AST abstract syntax tree 112

BDG bipartite directed graph 73

BFS breadth-first search 88

CLS combinatory logic synthesis 46

CRDT convergent or commutative replicated data type 101

CSP communicating sequential processes 21

DAG direct acyclic graph 81

DFS depth-first search . . . 88

DGML directed graph markup language . . . 112

DSL domain-specific language . . . 110

DTM deterministic Turing machine . . . 94

EBNF extended Backus-Naur form . . . 112

eDSL embedded domain-specific language . . . 110

EF entity framework . . . 163

ERP enterprise resource planning . . . 11

FSM finite state machine . . . 30

HPC high performance computing . . . 89

IIS internet information server . . . 179

IM interconnection model . . . 29

LLOC logical lines of code . . . 11

LTL linear time logic . . . 47

MEF managed extensibility framework . . . 162

MOF meta-object facility 148

NTM non deterministic Turing machine 94

OCL object constraint language 148

OMG object management group 148

OWL-S ontology web language 145

PC procedure call 28

PDCA plan–do–check–act 192

PKI public key infrastructure 167

POCO plain old C# Objects 174

REST representational state transfer 133

RPC remote procedure call 28

RQS rolling queue strategy 88

SCS staged composition synthesis 148

SLD selective linear definite 46

SLTL semantic linear time logic 49

STRIPS Stanford Research Institute Problem Solver 49

SysML systems modeling language 21

T4 text template transformation toolkit 156

TCP term complexity partitioner 89

UML unified modeling language 21

UR utilization ratio 89

WCF windows communication foundation 13

WSDL web service description language 166

WSMO web service modeling ontology 145

XML extensible markup language 22

List of Figures

1.1 Interconnections between main contributions in the thesis . . . 16

2.1 I-connector . . . 24
2.2 Overview of the Combinatory Logic Connector Synthesis method. 26

3.1 Bounded combinatory logic $\mathsf{BCL}_k(\cap, \leq)$ . . . 41

4.1 Example proof graph with lookahead . . . 63
4.2 Logarithmic Plot of Ratio of Created Inhabitation Questions . 68
4.3 Execution Graph for $\Gamma \vdash ? : \tau$ . . . 74
4.4 Acceptance rule in $\mathcal{G}_E$ . . . 76
4.5 Example graph with multiple nodes . . . 81
4.6 Example graph (see for Figure 4.5 on page 81) with shared representation as DAG . . . 82
4.7 Execution Graph with cached result for $\Gamma \vdash ? : \tau'$ . . . 83
4.8 Execution Graph with a reoccuring $\Gamma \vdash ? : \tau'$ node . . . 83
4.9 Execution Graph with a reused $\Gamma \vdash ? : \tau'$ node . . . 84
4.10 Execution Graph containing a cyclic solution . . . 85
4.11 Execution Graph containing a cyclic solution with a backward edge . . . 86
4.12 Execution Graph marked with success containing a cyclic solution with a backward edge . . . 87
4.13 Plot of Experimentally measured $\overline{U_R}$ for RQS . . . 91
4.14 Plot of Experimentally measured $\overline{U_R}$ for TCP . . . 91

5.1 Constructing inhabitant from an execution graph $\mathcal{G}_E$ . . . 106
5.2 (CL)S architecture as layer diagram . . . 108
5.3 (CL)S package diagram . . . 109
5.4 (CL)S inter-object dependencies . . . 110
5.5 (CL)S execution graph in DGML . . . 113
5.6 (CL)S execution graph in DGML . . . 114
5.7 (CL)S support in the open source programming editor `Notepad++` 116

5.8 SCS support in Microsoft Visual Studio 2013 116
5.9 SCS support in the open source programming editor Notepad++ 117

6.1 Atomic building block 123
6.2 Complex building block 123
6.3 Container building block 125
6.4 Adapter building block 126
6.5 Example (abstract) taxonomy 127
6.6 Example meta-taxonomy 127
6.7 Connector classes taxonomy inspired by Hirsch et al. [1999] . . 130
6.8 Example connector type Arbitrator from Taylor et al. [2010, pages 166ff] 131
6.9 Example connector type Arbitrator from Taylor et al. [2010, pages 171f] 132
6.10 Connector compatibility matrix from Taylor et al. [2010, pages 178f] 133
6.11 Excerpt of a connector taxonomy included in ArchiType 134
6.12 Components A and B 136
6.13 Components S and C 137
6.14 Building block $\texttt{ws}_{enc}$ 137
6.15 Resulting connector $\texttt{ws}_{enc}(\mathsf{C})$ 138
6.16 Resulting connector $\texttt{ws}_{enc}(\mathsf{C})$ with components A and B 138

7.1 Specification of ArchiType specific stereotypes for UML2 elements 150
7.2 Generation process from inhabitant to UML2 connector 151
7.3 ArchiType screenshot before generation 152
7.4 ArchiType screenshot after generation 153
7.5 Generation process from UML2 to source code 153
7.6 Three levels of computation in SCS 159
7.7 UML2 object diagram for ArchiType 163

8.1 Generated code for classA in secure connectors scenario . . . 168
8.2 Generated code for service host Server in secure connectors scenario 168
8.3 Many-to-many client server application 171
8.4 Many-to-many client server application with broker pattern [Buschmann et al., 1996, pages 237ff] 172
8.5 Visualization of XERP.NET's client's subsystem dependencies 177
8.6 Visualization of XERP.NET's server's subsystem dependencies 178
8.7 Visualization of AspxCommerce's subsystem dependencies . . 181
8.8 Generated UML2-component diagram in VS2013 for component CurrencyConverter 182

8.9 Screenshot of the configuration interface of the *internet information server* (IIS) hosting AspxCommerce visible in the background 183
8.10 Screenshot of AspxCommerce including generated code 184

List of Tables

3.1 Complexity results for various provability (inhabitation) problems 34

4.1 Experimental Data for Γ_n^m 69
4.2 Experimentally measured $\overline{U_R}$ for RQS 90
4.3 Experimentally measured speedup for RQS 92
4.4 Experimentally measured $\overline{U_R}$ for TCP 92
4.5 Experimentally measured speedup for TCP 92

5.1 Mathematical operators and corresponding expressions in (CL)S 111

6.1 Connector classes and instances according to Hirsch et al. [1999] 129

8.1 Experimental results for various applications of ArchiType . . . 189

B.1 Problem definitions used for the experiments in the Subsection 4.2.4 on page 87 208

List of Algorithms

3.1 ATM deciding inhabitation in $\mathsf{BCL}_0(\cap, \leq)$ 44

4.1 ATM deciding inhabitation in $\mathsf{BCL}_k(\cap, \leq)$ 53
4.2 `INH1'`(Γ, τ, k) 56
4.3 `Match`(C) from Düdder et al. [2013a] 58
4.4 `INH2`(Γ, τ, k) 60
4.5 ATM with lookahead-test 64
4.6 Alternating Turing Machine deciding inhabitation in $\mathsf{BCL}_k(\cap, \leq)$ with marked choices 72
4.7 Algorithm `InhabOptimized` for (CL)S 96
4.8 Sequential Control Algorithm for (CL)S 97
4.9 Concurrent Control Algorithm for (CL)S 102

5.1 Algorithm for reconstruction of inhabitants `Reconstruct` 107

Bibliography

Johnathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of ICSE'02*, pages 187–197. ACM, 2002. (Cited on page 164)

Jonathan Aldrich. Using Types to Enforce Architectural Structure. In *WICSA*, pages 211–220. IEEE Computer Society, 2008. (Cited on pages 139 and 164)

Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, January 1997. Issued as CMU Technical Report CMU-CS-97-144. (Cited on page 21)

Robert J. Allen and David Garlan. Formalizing Architectural Connection. In Bruno Fadini, Leon J. Osterweil, and Axel van Lamsweerde, editors, *ICSE*, pages 71–80. IEEE Computer Society / ACM Press, 1994. ISBN 0-8186-5855-X. (Cited on page 143)

Robert J. Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997. (Cited on pages 11 and 119)

Marco Autili, Chris Chilton, Paola Inverardi, Marta Kwiatkowska, and Massimo Tivoli. Towards a Connector Algebra. In *Proceedings of ISoLA'10*, volume 6416 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2010. (Cited on page 144)

Lerina Aversano, Massimiliano Di Penta, and Kunal Taneja. A Genetic Programming Approach to Support the Design of Service Compositions. *Computer Systems Science and Engineering*, 21(4), 2006. (Cited on page 145)

Hamid Bagheri. A Formal Approach to Software Synthesis for Architectural Platforms. In *Proceedings of ICSE'11*, pages 1143–1145. ACM, 2011. (Cited on page 164)

Dusan Balek. *Connectors in Software Architectures*. PhD thesis, Charles University, Czech Republic, 2002. (Cited on page 27)

Dusan Balek and Frantisek Plasil. Software Connectors and Their Role in Component Deployment. In *Proceedings of DAIS'01*, Krakow, Poland, September 2001. Kluwer Academic Publishers. (Cited on page 27)

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983. (Cited on pages 5, 9, 35, 36, 37, 38, 49, 66, 78, 110, and 120)

Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2013. (Cited on page 36)

David A. Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson. Synthesis of Programs in Computational Logic. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 30–65. Springer, 2004. ISBN 3-540-22152-2. (Cited on page 47)

Len Bass, Paul C. Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003. (Cited on pages 20 and 21)

Amel Bennaceur, Chris Chilton, Malte Isberner, and Bengt Jonsson. Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 274–288. Springer, 2013. ISBN 978-3-642-40560-0. (Cited on page 144)

Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting Families of Software Systems with Process Algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002. (Cited on page 22)

Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of Web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3–4):316–361, 2010. (Cited on page 145)

Jan Bessai, Andrej Dudenhefner, Boris Düdder, and Moritz Martens. Delegation-based Mixin Composition Synthesis. In Jakob Rehof, editor, *Proceedings of Intersection Types and Related Systems (ITRS'14)*, Lecture Notes in Computer Science. Springer, 2014a. (Cited on pages 14 and 115)

Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. Combinatory Logic Synthesizer. In Bernhard Steffen, editor, *Proceedings of the 6th International Symposium On Leveraging Applications of*

Formal Methods, Verification and Validation (ISoLA'14), Lecture Notes in Computer Science. Springer, 2014b. (Cited on pages 14 and 115)

Robert D. Blumofe. Scheduling Multithreaded Computations by Work Stealing. In *FOCS*, pages 356–368. IEEE Computer Society, 1994. (Cited on page 101)

Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, 1999. (Cited on page 101)

Rastislav Bodik and Barbara Jobstmann. Algorithmic Program Synthesis: Introduction. *International Journal on Software Tools for Technology Transfer*, 15(5-6):397–411, 2013. ISSN 1433-2779. (Cited on page 47)

Robert S. Boyer and J. Strother Moore. *The Sharing of Structure in Theorem-proving Programs*, volume 7 of *Machine Intelligence*, pages 101–116. Edinburgh University Press, 1972. (Cited on page 81)

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. John Wiley and Sons, 1996. (Cited on pages 29, 100, 101, 125, 144, 170, 172, 173, and 216)

Bradford L. Chamberlain. Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations. Technical Report UW-CSE-98-10-03, University of Washington, 1998. (Cited on page 89)

Ashok K. Chandra, Dexter. C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981. (Cited on pages 43, 45, and 95)

Christina Chavez, Alessandro F. Garcia, Thaís Vasconcelos Batista, Marcel Vinicius Medeiros Oliveira, Cláudio Sant'Anna, and Awais Rashid. Composing architectural aspects based on style semantics. In Kevin J. Sullivan, Ana Moreira, Christa Schwanninger, and Jeff Gray, editors, *AOSD*, pages 111–122. ACM, 2009. ISBN 978-1-60558-442-3. (Cited on page 31)

Alonzo Church. Applications of Recursive Arithmetic to the Problem of Circuit Synthesis. *Summaries of the Summer Institute of Symbolic Logic*, I: 3 – 50, 1957. (Cited on page 47)

Paul C. Clements. A Survey of Architecture Description Languages. In *Proc. Int'l Workshop on Software Specification and Design*, pages 16–25. IEEE Press, March 1996. (Cited on page 21)

Paul C. Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2011. ISBN 0201703726. (Cited on pages 21 and 22)

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and Lattices for Distributed Programming. In Michael J. Carey and Steven Hand, editors, *SoCC*, page 1. ACM, 2012. ISBN 978-1-4503-1761-0. (Cited on page 198)

Andy Coombes and John McDermid. A Tool for Defining the Architecture of Z Specifications. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 77–92. Springer London, 1991. ISBN 978-3-540-19672-3. (Cited on page 143)

Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. In Alfred V. Aho, editor, *STOC*, pages 1–6. ACM, 1987. ISBN 0-89791-221-7. (Cited on page 78)

Mario Coppo and Mariangiola Dezani-Ciancaglini. An Extension of Basic Functionality Theory for Lambda-Calculus. *Notre Dame Journal of Formal Logic*, 21:685–693, 1980. (Cited on page 35)

Krzysztof Czarnecki. Overview of Generative Software Development. In *In Proceedings of Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France, Revised Papers*, pages 313–328. Springer-Verlag, 2004. (Cited on page 191)

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Application*. Addison-Wesley, Reading, MA, 2000. (Cited on page 191)

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001. (Cited on page 141)

Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 266–276. ACM Press, 2002. ISBN 1-58113-472-X. (Cited on page 22)

Rowan Davies and Frank Pfenning. A Modal Analysis of Staged Computation. *Journal of the ACM*, 48(3):555–604, 2001. (Cited on page 160)

Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Proceedings of ESEC/FSE'01*, pages 109–120. ACM, 2001. (Cited on page 144)

William Edwards Deming. *Out of the Crisis*. The MIT Press, 2000. (Cited on page 192)

Mariangiola Dezani-Ciancaglini and J. Roger Hindley. Intersection Types for Combinatory Logic. *Theoretical Computer Science*, 100(2):303–324, 1992. (Cited on pages 5, 34, and 35)

Boris Düdder. Formale Verifikation mittels Model Checking in Materialflusssystemen. Diplomarbeit, Universität Dortmund, Fakultät für Informatik, 2008. Interne Berichte / Universität Dortmund. (Cited on page 14)

Boris Düdder, Guido Follert, and Moritz Roidl. Model Checking in multiagentengesteuerten Materialflusssystemen. In Peter Buchoholz, editor, *Workshop "Modellierung großer Netze in der Logistik", Measurement, Modelling and Evaluation of Computer and Communication Systems [Tagung Dortmund 1. April 2008]*, volume 817 of *Technical Reports*. Technical University Dortmund, 2008. (Cited on page 14)

Boris Düdder, Oliver Garbe, Moritz Martens, Jakob Rehof, and PawełUrzyczyn. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for GUI Synthesis. In *Proceedings of ITRS'12*, 2012. (Cited on pages 10, 14, 48, 49, 52, 135, and 141)

Boris Düdder, Moritz Martens, and Jakob Rehof. Intersection Type Matching and Bounded Combinatory Logic (Extended Version). Technical Report 841, Department of Computer Science (TU Dortmund), 2012. `http://www-seal.cs.tu-dortmund.de/seal/downloads/research/cls/TR841-TypeMatching.pdf`. (Cited on page 63)

Boris Düdder, Moritz Martens, Jakob Rehof, and PawełUrzyczyn. Bounded Combinatory Logic. In *Proceedings of CSL'12*, volume 16 of *LIPIcs*, pages 243–258. Schloss Dagstuhl, 2012. (Cited on pages 10, 14, 34, 35, 39, 44, 45, 48, 49, 52, 53, 55, 56, 139, and 142)

Boris Düdder, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn. Bounded Combinatory Logic (Extended Version). Technical Report 840, Department of Computer Science (TU Dortmund), 2012. `http://www-seal.cs.tu-dortmund.de/seal/downloads/research/cls/TR_840-BCL.pdf`. (Cited on pages 35 and 55)

Boris Düdder, Moritz Martens, Jakob Rehof, and PawełUrzyczyn. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Synthesis. Technical Report 842, Department of Computer Science (TU Dortmund), 2012. `http://www-seal.cs.tu-dortmund.de/seal/downloads/research/cls/TR842-ITRS.pdf`. (Cited on page 52)

Boris Düdder, Moritz Martens, and Jakob Rehof. Intersection Type Matching with Subtyping. In *Proceedings of TLCA'13*, volume 7941 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2013a. (Cited on pages 10, 14, 49, 57, 58, 59, 61, 131, and 221)

Boris Düdder, Moritz Martens, and Jakob Rehof. A Theory of Staged Composition Synthesis (Extended Version). Technical Report 843, Department of Computer Science (TU Dortmund), 2013b. `http://www-seal.cs.tu-dortmund.de/seal/downloads/research/cls/TR843-SCS.pdf`. (Cited on pages 112 and 159)

Boris Düdder, Moritz Martens, and Jakob Rehof. Staged Computation Synthesis. In Zhong Shao, editor, *Proceedings of 23rd European Symposium on Programming (ESOP'14)*, volume 8410 of *Lecture Notes in Computer Science*, pages 67–86. Springer, 2014a. (Cited on pages 14, 112, 115, 159, and 160)

Boris Düdder, Moritz Martens, and Jakob Rehof. Combinatory Process Synthesis. In Radu Mateescu, editor, *Proceedings of 12th International Conference on Software Engineering and Formal Methods (SEFM'14)*, Lecture Notes in Computer Science. Springer, 2014b. (Cited on pages 115 and 135)

José Luiz Fiadeiro, Antónia Lopes, and Michel Wermelinger. A Mathematical Semantics for Architectural Connectors. In Roland Carl Backhouse and Jeremy Gibbons, editors, *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 178–221. Springer, 2003. ISBN 3-540-20194-7. (Cited on page 144)

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. (Cited on page 133)

Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of PLDI'91*, pages 268–277. ACM, 1991. (Cited on page 48)

Dadid Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering*

and Knowledge Engineering, volume I. River Edge, NJ: World Scientific Publishing Company, 1993. (Cited on page 143)

David Garlan and Dewayne Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995. (Cited on page 20)

David Garlan, Robert T. Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997. (Cited on pages 21 and 31)

Simon Gay. ICT COST Action IC1201, Behavioural Types for Reliable Large-Scale Software Systems (BETTY). `http://www.cost.eu/domains_actions/ict/Actions/IC1201`, 2012. Accessed: 2014-02-25. (Cited on page 198)

Gregor Gößler and Joseph Sifakis. Composition for Component-based Modeling. *Sci. Comput. Program.*, 55(1–3):161–183, 2005. (Cited on page 31)

Benjamin N. Grossof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of WWW 2003*, May 2003. (Cited on page 49)

Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. (Cited on page 32)

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of PLDI'11*, pages 62–73. ACM, 2011. (Cited on page 48)

Christian Haack, Brian Howard, Allen Stoughton, and Joe B. Wells. Fully Automatic Adaptation of Software Components Based on Semantic Specifications. In *AMAST*, volume 2422 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2002. (Cited on pages 6, 9, 48, and 162)

George T. Heineman. A Model for Designing Adaptable Software Components. *ACM SIGSOFT Software Engineering Notes*, 25(1):55–56, 2000. (Cited on page 191)

George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, June 2001. (Cited on page 191)

Fritz Henglein. Dynamic Typing: Syntax and Proof Theory. *Sci. Comput. Program.*, 22(3):197–230, 1994. (Cited on page 198)

J. Roger Hindley. The Simple Semantics for Coppo-Dezani-Sallé Types. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 1982. (Cited on pages 38 and 77)

J. Roger Hindley and Jonathan P. Seldin. *Lambda-calculus and Combinators, an Introduction*. Cambridge University Press, 2008. (Cited on pages 4, 5, and 49)

Dan Hirsch, Sebastián Uchitel, and Daniel Yankelevich. Towards a Periodic Table of Connectors. In *Proceedings of COORDINATION '99*, volume 1594 of *Lecture Notes in Computer Science*. Springer, 1999. (Cited on pages 9, 11, 25, 32, 128, 129, 130, 175, 216, and 219)

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1985. (Cited on page 22)

IEEE Architecture Working Group. IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000. (Cited on page 19)

Paola Inverardi and Massimo Tivoli. Automatic Synthesis of Modular Connectors via Composition of Protocol Mediation Patterns. In *Proceedings of ICSE'13*, pages 3–12. IEEE, 2013. (Cited on pages 32 and 144)

ISO/IEC/IEEE. Systems and software engineering – Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1 –46, 1 2011. (Cited on page 21)

Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *TOSEM*, 11(2):256–290, 2002. (Cited on pages 143 and 164)

Dietmar Jannach and Klaus Leopold. Knowledge-based multimedia adaptation for ubiquitous multimedia consumption. *J. Network and Computer Applications*, 30(3):958–982, 2007. (Cited on page 49)

Neil D. Jones. An Introduction to Partial Evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996. (Cited on page 161)

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN 978-0-13-020249-9. (Cited on page 161)

Christine Julien and Dewayne Perry. Composable Context-aware Architectural Connectors. In *Proceedings of SAM '08*, pages 43–45. ACM, 2008. (Cited on pages 24 and 144)

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Combinators for Impure yet Hygienic Code Generation. In Wei-Ngan Chin and Jurriaan Hage, editors, *ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM 14)*, pages 3–14. ACM, 2014. (Cited on page 191)

Raman Kazhamiakin, Annapaola Marconi, Marco Pistore, and Heorhi Raik. Data-Flow Requirements for Dynamic Service Composition. In *International Conference on Web Services (ICWS)*, pages 243–250. IEEE, 2013. (Cited on page 145)

Stephen Kell. Rethinking Software Connectors. In *Proceedings of SYANCO '07*, pages 1–12. ACM, 2007. (Cited on pages 24 and 27)

Philippe Kruchten. An Ontology of Architectural Design Decisions in Software-Intensive Systems. In *Proceedings of the 2nd Groningen Workshop on Software Variability Managment*, 2004. (Cited on pages 9, 31, and 32)

Patrick Lam and Martin C. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 275–302. Springer, 2003. ISBN 3-540-40531-3. (Cited on page 141)

Anna-Lena Lamprecht, Tiziana Margaria, Ina Schaefer, and Bernhard Steffen. Synthesis-Based Variability Control: Correctness by Construction. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*, volume 7542 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011. ISBN 978-3-642-35886-9, 978-3-642-35887-6. (Cited on page 49)

Martin Lange and Carsten Lutz. 2-ExpTime lower bounds for propositional dynamic logics with intersection. *Journal of Symbolic Logic*, 70(4):1072–1086, 2005. (Cited on page 48)

Kung-Kiu Lau, Ling Ling, Vladyslav Ukis, and Perla Velasco Elizondo. Composite Connectors for Composing Software Components. In Markus Lumpe

and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2007. ISBN 978-3-540-77350-4. (Cited on page 144)

Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency without Concurrency Control. *CoRR*, abs/0907.0929, 2009. (Cited on page 103)

Samuel Linial and Emil L. Post. Recursive Unsolvability of the Deducibility, Tarski's Completeness and Independence of Axioms Problems of Propositional Calculus. *Bulletin of the American Mathematical Society*, 55:50, 1949. (Cited on pages 5 and 34)

Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.*, 16:1811–1841, November 1994. ISSN 0164-0925. (Cited on page 193)

David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Walter Mann, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. (Cited on page 21)

Yoad Lustig and Moshe Y. Vardi. Synthesis from Component Libraries. In *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2009. (Cited on pages 3, 13, and 47)

Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proceedings of ESEC '95 – 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, Sitges, Spain, 25–28 September 1995. Springer-Verlag, Berlin, Germany. (Cited on page 22)

Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *Software Engineering, IEEE Transactions on*, 39(6):869–891, 2013. (Cited on page 21)

Zohar Manna and Richard Waldinger. Fundamentals Of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, 18:674–704, 1992. (Cited on page 3)

Zohar Manna and Richard J. Waldinger. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980. (Cited on page 3)

Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In *Proceedings of FSE'13*, pages 444–454. ACM, 2013. (Cited on pages 143 and 144)

Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002. ISBN 0135974445. (Cited on page 193)

Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Engineering*, 26(1):70–93, 2000. ISSN 0098-5589. (Cited on page 21)

Nenad Medvidovic, Peyman Oreizy, and Richard N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In W. Richards Adrion, Alfonso Fuggetta, Richard N. Taylor, and Anthony I. Wasserman, editors, *ICSE*, pages 692–700. ACM, 1997. ISBN 0-89791-914-9. (Cited on page 21)

Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002. (Cited on page 22)

Nikunj R. Mehta and Nenad Medvidovic. Composing architectural styles from architectural primitives. In *ESEC / SIGSOFT FSE*, pages 347–350. ACM, 2003. (Cited on page 30)

Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of ICSE'00*, pages 178–187. ACM, 2000. (Cited on pages 9, 11, 25, 27, 32, 130, and 175)

Bertrand Meyer. *Object-Oriented Software Construction*. International Series in Computer Science, C.A.R. Hoare, Series Editor. Prentice Hall, 1988. (Cited on pages 190 and 193)

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Ann. Pure Appl. Logic*, 51 (1–2):125–157, 1991. (Cited on pages 47 and 141)

Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980. (Cited on page 22)

Robin Milner. *Communication and Concurrency*. PHI Series in computer science. Prentice Hall, 1989. (Cited on page 22)

Robin Milner. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press, 1999. ISBN 978-0-521-65869-0. (Cited on page 22)

J Strother Moore. *Computational Logic: Structure Sharing and Proof of Program Properties.* PhD thesis, Department of Computational Logic, University of Edinburgh, 1973. (Cited on page 81)

Mark Moriconi and Xiaolei Qian. Correctness and Composition of Software Architectures. In *Proceedings of FSE'94*, pages 164–174. ACM, 1994. (Cited on page 143)

Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog.* Wiley, 1990. ISBN 978-0-471-92625-2. (Cited on page 141)

Object Management Group (OMG). Unified Modeling Language (UML), V2.0. `http://www.omg.org/spec/UML/Current`, 2005. Accessed: 2013-08-16. (Cited on pages 21, 124, and 147)

Object Management Group (OMG). MetaObject Facility (MOF), V2.4.1. `http://www.omg.org/spec/MOF/`, 2011. Accessed: 2014-01-06. (Cited on page 148)

Object Management Group (OMG). Systems Modeling Language (SysML), V1.3. `http://www.omgsysml.org`, 2012. Accessed: 2013-12-27. (Cited on page 21)

Flavio Oquendo. pi-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004. (Cited on page 22)

Flávio Oquendo. Dynamic Software Architectures: Formally Modelling Structure and Behaviour with π-ADL. In *ICSEA*, pages 352–359. IEEE Computer Society, 2008. (Cited on page 22)

Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994. (Cited on pages 43 and 94)

John Penix. Deductive Synthesis of Event-Based Software Architectures. In *Proceedings of the ASE'99*, pages 311–314, 1999. (Cited on page 144)

Dewayne E. Perry. Software Interconnection Models. In William E. Riddle, Robert M. Balzer, and Kouichi Kishida, editors, *ICSE*, pages 61–71. ACM Press, 1987. ISBN 0-89791-216-0. (Cited on page 29)

Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992. (Cited on pages 11, 20, and 119)

Andrew Pitts and Tim Sheard. On the Denotational Semantics of Staged Execution of Open Code. In *Nineteenth Annual IEEE Symposium On Logic In Computer Science*, pages 1–16. Springer Verlag, 2004. (Cited on page 191)

Steffen Plate. Automatische Generierung einer Konfiguration für virtuelle Maschinen unter Zuhilfenahme eines Inhabitationsalgorithmus. Bachelor's thesis, Technical University of Dortmund, Department of Computer Science, 2013. (Cited on page 115)

Garrel Pottinger. A Type Assignment for the Strongly Normalizable Lambda-Terms. In J. Roger Hindley and Johnathan P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980. (Cited on page 35)

Vaughan Pratt and Jerzy Tiuryn. Satisfiability of Inequalities in a Poset. *Fundamenta Informaticae*, 28(1–2):165–182, 1996. (Cited on page 78)

Jorge Enrique Pérez-Martínez and Almudena Sierra-Alonso. UML 1.4 versus UML 2.0 as Languages to Describe Software Architectures. In Flavio Oquendo, Brian C. Warboys, and Ron Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 88–102. Springer Berlin Heidelberg, 2004. (Cited on page 22)

Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In Jorge Cardoso and Amit P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2004. (Cited on page 145)

Jakob Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, Department of Computer Science, 1998. (Cited on page 78)

Jakob Rehof. Towards Combinatory Logic Synthesis. In *BEAT'13, 1st International Workshop on Behavioural Types*. ACM, January 22 2013. (Cited on pages 3, 5, 7, 9, 10, 46, 48, 49, 112, 126, 138, 207, and 208)

Jakob Rehof and Torben Mogensen. Tractable Constraints in Finite Semilattices. *Science of Computer Programming*, 35(2):191–221, 1999. (Cited on page 78)

Jakob Rehof and Paweł Urzyczyn. Finite Combinatory Logic with Intersection Types. In *Proceedings of TLCA'11*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011a. (Cited on pages 9, 10, 34, 36, 37, 41, 48, 49, 52, 78, 84, 87, and 169)

Jakob Rehof and Paweł Urzyczyn. Finite Combinatory Logic with Intersection Types (Extended Version). Technical Report 834, Department of Computer Science (TU Dortmund), 2011b. (Cited on page 87)

Jakob Rehof and Paweł Urzyczyn. The Complexity of Inhabitation with Explicit Intersection. In *Kozen Festschrift*, volume 7230 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2012. (Cited on pages 10, 34, 38, 48, and 142)

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid Types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008. (Cited on page 198)

Sylvain Salvati. Recognizability in the Simply Typed Lambda-Calculus. In H. Ono, M. Kanazawa, and R. J. G. B. de Queiroz, editors, *WoLLIC*, volume 5514 of *Lecture Notes in Computer Science*, pages 48–60. Springer, 2009. (Cited on page 35)

Sylvain Salvati, Giulio Manzonetto, Mai Gehrke, and Henk Barendregt. Loader and Urzyczyn Are Logically Related. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *ICALP (2)*, volume 7392 of *Lecture Notes in Computer Science*, pages 364–376. Springer, 2012. ISBN 978-3-642-31584-8. (Cited on page 35)

Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 2 edition, 1995. (Cited on pages 137 and 169)

Ehud Y. Shapiro. Alternation and the computational complexity of logic programs. *J. Log. Program.*, 1(1):19–33, 1984. (Cited on pages 73 and 75)

Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and Commutative Replicated Data Types. *Bulletin of the EATCS*, 104:67–88, 2011. (Cited on page 101)

Mary Shaw and David Garlan. *Software Architecture - Perspectives on an emerging discipline*. Prentice Hall, 1996. (Cited on page 19)

Joseph Sifakis. A Framework for Component-based Construction Extended Abstract. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 293–300. IEEE Computer Society, 2005. (Cited on page 31)

Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2nd edition, August 2008. ISBN 1848000693. (Cited on page 78)

Morten H. Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006. (Cited on page 4)

Romina Spalazzese and Paola Inverardi. Components Interoperability through Mediating Connector Patterns. In Javier Cámara, Carlos Canal, and Gwen Salaün, editors, *WCSI*, volume 37 of *EPTCS*, pages 27–41, 2010. (Cited on page 32)

Bridget Spitznagel and David Garlan. A Compositional Approach for Constructing Connectors. In *Proceedings of WICSA'01*, pages 148–157. IEEE, 2001. (Cited on pages 11, 24, 125, 140, and 144)

Bridget Spitznagel and David Garlan. A Compositional Formalization of Connector Wrappers. In *Proceedings of ICSE'03*, pages 374–384. IEEE, 2003. (Cited on page 11)

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From Program Verification to Program Synthesis. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 313–326. ACM, 2010. ISBN 978-1-60558-479-9. (Cited on page 48)

Richard Statman. Intuitionistic Propositional Logic Is Polynomial-space Complete. *Theoretical Computer Science*, 9:67–72, 1979. (Cited on pages 4, 33, 34, and 35)

Bernhard Steffen, Tiziana Margaria, and Michael von der Beeck. Automatic Synthesis of Linear Process Models from Temporal Constraints: An Incremental Approach. In *In ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97)*, 1997. (Cited on pages 49 and 126)

Andrew James Stothers. *On the Complexity of Matrix Multiplication*. University of Edinburgh, 2010. (Cited on page 78)

Misha Strittmatter and Lucia Kapová Happe. Compositional performance abstractions of software connectors. In David R. Kaeli, Jerry Rolia, Lizy K. John, and Diwakar Krishnamurthy, editors, *ICPE*, pages 275–278. ACM, 2012. ISBN 978-1-4503-1202-8. (Cited on page 31)

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1–2):211–242, 2000. (Cited on page 191)

Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. Addison-Wesley, 2010. (Cited on pages 11, 20, 21, 22, 23, 26, 27, 28, 29, 32, 119, 126, 130, 131, 132, 133, 140, 144, and 216)

Jerzy Tiuryn. Subtype Inequalities. In *Proceedings of LICS'92*, pages 308–315. IEEE Computer Society, 1992. (Cited on page 78)

Jonathan Traugott. Deductive Synthesis of Sorting Programs. *J. Symb. Comput.*, 7(6):533–572, 1989. (Cited on page 3)

Heikki Tuominen. Elementary net systems and dynamic logic. In Grzegorz Rozenberg, editor, *European Workshop on Applications and Theory in Petri Nets*, volume 424 of *Lecture Notes in Computer Science*, pages 453–466. Springer, 1988. ISBN 3-540-52494-0. (Cited on page 48)

Paweł Urzyczyn. The Emptiness Problem for Intersection Types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999. (Cited on pages 33, 34, and 35)

Paweł Urzyczyn. Inhabitation of Low-Rank Intersection Types. In *Proceedings of TLCA'09*, volume 5608 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2009. (Cited on page 34)

Moshe Y. Vardi. From Verification to Synthesis. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, page 2. Springer, 2008. ISBN 978-3-540-87872-8. (Cited on page 48)

Anna Vasileva. Synthese von Orchestrationscode für Cloud-basierte Dienste. Diploma thesis, Technical University of Dortmund, Department of Computer Science, 2013. (Cited on page 115)

Richard J. Waldinger. Tutorial on Program-Synthetic Deduction. In Mark E. Stickel, editor, *CADE*, volume 449 of *Lecture Notes in Computer Science*, page 684. Springer, 1990. (Cited on page 3)

Chris Walshaw, Mark Cross, and Martin G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997. (Cited on page 89)

Stephen Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9 (1):11–12, 1962. (Cited on page 79)

Robert Waters and Gregory D. Abowd. Architectural Synthesis: Integrating Multiple Architectural Perspectives. In *WCRE*, pages 2–12, 1999. (Cited on page 145)

Joe B. Wells and Boris Yakobowski. Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis. In *LOPSTR 2004*, volume 3573 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2005. (Cited on pages 6, 9, and 48)

Patrick Wolf. Entwicklung einer Adapters mit VI Scripting (LabVIEW) zur Synthese von LEGO® NXT-VIs aus einem Repository. Bachelor's thesis, Technical University of Dortmund, Department of Computer Science, 2013. (Cited on page 115)

Daniel Yellin and Robert E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19 (2):292–333, 1997. (Cited on page 144)

Index

⌀MQ, 108

ADL, 21
  π-ADL, 22
  AADL, 21
  Acme, 21
  Darwin, 22
  PADL, 22
  Rapide, 21
  Wright, 21
  xADL, 22
Algorithm
  Coppersmith&Winograd, 78
  Warshalls, 79
Alloy, 164
Applicability, 190
Architecture
  Communication integrity, 164
  Component, 23
  Conformance, 139
  Connector, 23
  Continuous improvement, 192
  Description Language, *see* ADL
  Evolution, 191
  Pattern
    Adapter, 29
    Broker, 170
  Style, 20
    Representational State Transfer, 134
ArchiType, 147
  Code generation, 152
  Composition Language, 154
  Implementation, 162
ArchJava, 164

Building block, 122
  Adapter, 124
  Atomic, 122
  Classification, 122
  Complex, 123
  Container, 123, 151

C&C Architecture, 22
C&C type environment, 126, 152
Calculus
  $\lambda_e^{\Box\rightarrow}$, 160
  π, 22
  CSP, 21
CALM theorem, 198
Combinatory Logic, 33
  Bounded, 33, 39
  Finite, 34, 41
Combinatory Logic Connector Synthesis, 25, 136
  Connector Synthesis Goal, 134
  Generation, 25, 135
  Specification, 25, 139
  Synthesis, 25, 134
  Synthesize and Generate, 136
Commutative replicated data type, 101, 108, 198
Component-based development, 13
Components

Adaptable, 191
Concurreny, 100
Connector type, 120
Control-centric, 72

Data-centric, 71

Execution graph
Group node, 73
Inhabitation node, 73
Export
Direct Graph Markup Language, 112
DOT, 112
Expressiveness, 189

Finite
Function, 142
Quantification, 143

Generation, 135

Hygienic code generation, 191

I-connector, 23
Inhabitants reconstruction, 105
Inhabitation, 41
Direct, 43
Inhabitation node
Child, 73
Failed, 74
Parent, 73
Successful, 74
Inhabitation problem
Normal, 4, 41
Relativized, 41
Interconnection, 29
Models, 29
Semantic, 30
Syntactic, 30
Unit, 29
Interface automata, 144
Interface type, 23

Limitations, 193
Logic programming, 46

Meta-programming, 191
MetaML, 191
Microsoft
T4 Text Templating, 148, 156
Visual Studio, 147
Windows Communication Foundation, 148, 166
Model-driven engineering, 148

Normalization, 77

Ontology, 31, 32
Optimization
Atomic substitution, 79
Atomic subtyping relation, 78
Bounded substitutions, 79
Caches, 80
Concurrency, 100
Cycle detection, 84
Execution Graph, 72
Graph compression, 81
Normalization, 77
Parallel computation, 87
Transitive closure on subtypes, 78
Type environment organization, 79

Packaging component, 151
Partial evaluation, 161
Polymorphism
Implicit, 34
Typical ambiguity, 34
Principle
Interface segregation, 194
Liskov substitution, 193
Open/closed, 193
Single responsibility, 193
SOLID, 194
Prolog, 46, 73

Protocol
REST, 107, 134
WS-SOAP, 107, 134, 137

Queue, 88
Work-stealing, 101

Ramp-up Costs, 191
Repository, see type environment46
Resudialization, 161

Scattering, 89
Scheduler, 88
Rolling Queue Strategy, 88
Term Complexity Partitioner, 89
Security
Authentication, 169
Encryption, 137, 166
Kerberos, 169
NT LAN Manager, 169
Public key infrastructure, 167
Semaphore, 100
Shared memory, 101
Signal, 100
SLD resolution, 46
Software architecture, 19
Definition of, 19
IEEE1471-2000, 19
Software connector, 26
Communicator, 28
Converter, 29
Coordinator, 28
Faciliator, 29
Roles, 28
Software engineering
Component-based, 191
Synthesis, 47
AI Planning, 49, 145
Candidate space, 47
Church-style, 12
Combinatory Logic, 5, 46
Combinatory Process, 115
Component-based, 12
Composition, 3
Curry-style, 12
Deductive, 3
Functional, 47
Mixins, 14, 115
Reactive, 47
SLTL, 49
Staged Computation, 115, 159
Temporal, 47
Web service composition, 145
Semantic, 145

Taxonomy, 31, 126
Tree, 126
Thread-safe, 101
Turing machine
Alternating, 43
Acceptance, 43
Eventual acceptance, 43
Deterministic, 94
Non-deterministic, 94
Type
Applicative term, 40
Assignment, 40
Assumptions, 40
Behavioral, 193
Dynamic typing, 198
Environment, 40
Higher-order, 139, 173
Implementation type correctness, 6, 161
Inhabitation, 3, 4, 34, 44
Intersection, 35
Kind, 39, 70, 79, 124, 125
Levels, 40
Liquid, 198
Matching, 57, 59
Modal, 159
Organized, 38, 77
Path, 38

Refinement, 48
Rules, 41
Simple, 40
Substitution, 34, 39, 40, 79
Level-k, 40
Subtyping, 36, 49, 78
Unique specification, 87

UML, 21
Component diagram, 147
Meta-Object Facility, 148
Packaging component, 147
Stereotypes, 148
Utilization, 90
Ratio, 90

Watchdog, 100
Web Service Description Language, 166
Worker threads, 100