A Lightweight Framework for Universal Fragment Composition

— with an application in the Semantic Web

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

MSc. Jakob Henriksson
geboren am 11.08.1979 in Norrköping, Schweden

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)
Prof. Michael Schröder
(Biotechnologisches Zentrum der TU Dresden)
Prof. Welf Löwe
(Växjö University, Schweden)

Tag der Verteidigung: Dresden, den 19. Dezember 2008

Dresden, den 14. Oktober 2008

Abstract

Domain-specific languages (DSLs) are useful tools for coping with complexity in software development. DSLs provide developers with appropriate constructs for specifying and solving the problems they are faced with. While the exact definition of DSLs can vary, they can roughly be divided into two categories: *embedded* and *non-embedded*. Embedded DSLs (E-DSLs) are integrated into general-purpose *host* languages (e.g. Java), while non-embedded DSLs (NE-DSLs) are standalone languages with their own tooling (e.g. compilers or interpreters). NE-DSLs can for example be found on the Semantic Web where they are used for querying or describing shared domain models (ontologies). A common theme with DSLs is naturally their support of *focused* expressive power. However, in many cases they do not support non–domain-specific *component-oriented* constructs that can be useful for developers. Such constructs are standard in general-purpose languages (procedures, methods, packages, libraries etc.). While E-DSLs have access to such constructs via their host languages, NE-DSLs do not have this opportunity. Instead, to support such notions, each of these languages have to be extended and their tooling updated accordingly. Such modifications can be costly and must be done individually for each language. A solution method for one language cannot easily be reused for another. There currently exist no appropriate technology for tackling this problem in a general manner.

Apart from identifying the need for a general approach to address this issue, we extend existing composition technology to provide a language-inclusive solution. We build upon *fragment-based* composition techniques and make them applicable to arbitrary (context-free) languages. We call this process for the composition techniques' *universalization*. The techniques are called fragment-based since their view of components—reusable software units with interfaces—are pieces of source code that conform to an underlying (context-free) language *grammar*. The universalization process is *grammar-driven*: given a base language grammar and a description of the compositional needs wrt. the composition techniques, an adapted grammar is created that corresponds to the specified needs. The result is thus an adapted grammar that forms the foundation for allowing to define and compose the desired *fragments*. We further build upon this grammar-driven universalization approach to allow developers to define the non–domain-specific component-oriented constructs that are needed for NE-DSLs. Developers are able to define both what those constructs should be, and how they are to be interpreted (via composition). Thus, developers can effectively define language extensions and their semantics. This solution is presented in a framework that can be reused for different languages, even if their notion of 'components' differ.

To demonstrate the approach and show its applicability, we apply it to two Semantic Web related NE-DSLs that are in need of component-oriented constructs. We introduce *modules* to the rule-based Web query language Xcerpt and *role models* to the Web Ontology Language OWL.

Acknowledgments

First of all I would like to thank my supervisor Uwe Aßmann for inviting me to Dresden and for helping me along in the world of research. We have had much fun discussing fragment-based software composition and trying to fit in somewhere between the fields of traditional software engineering and the Semantic Web. I will not soon forget the early flights to Munich and our traditional "Weißwürste und Bier" breakfasts in preparation for project meetings. A big thanks to you Uwe, and the whole software technology group at TU Dresden, for having made me feel at home.

I owe a great debt to Jan Małuszyński, whom I had the great pleasure to work with. Jan has been a great mentor to me, the kind of mentor all young researchers should have. Jan has taught me much of what I know about writing and preparing research talks. I would also like to thank Włodek Drabent for our joint work, and from whom I also learned much about paper writing and critical thinking.

Having worked on this thesis topic in solitude would have been uninspiring and unrewarding. Jendrik Johannes was first a student of mine, and later a great friend and colleague. Jendrik has more than anyone been helpful during this work and I do not think it would have been done without him. Our software composition "séances" in the hidden cafés of Neustadt are fond memories. Steffen Zschaler has also been a fierce fragment composition proponent and has been of great value in forwarding this research. I've also very much enjoyed working with Florian Heidenreich – thank you for your support and friendship.

Much of this research was carried out in the REWERSE project. I want to mention Tim Furche, whom it always was a great pleasure to meet and work with. We worked together on investigating modules for the rule-based query language Xcerpt. I think we had a great collaboration, and I appreciated it very much. I also want to mention Sacha Berger who was a key player in this work. I had the privilege of supervising Michael Pradel, whom I worked with in investigating modularization techniques for ontology languages. I greatly thank him for that joint work.

I want to thank Ilie Savga, whom I shared an office with, and could share all thesis agony with. Thanks to Uwe, Ilie, Jendrik and Sven Karol for your invaluable comments on the written text. Any remaining errors and inconsistencies are naturally my own.

Finally, but not lastly, I want to thank my family - Jan-Erik, Barbro, Jonatan, Andreas and Aron. They encouraged and supported me throughout this whole experience. During this time I also started my own family by marrying Christina Hade. Without her love and support this work would never have been completed.

Jakob Henriksson
Dresden, October 2008

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme (FP) project REWERSE (No.: 506779 (cf. `http://rewerse.net`), and the 7th FP project MOST (No.: 216691, cf. `http://most-project.eu`).

Publications

This thesis is partially based on the following peer-reviewed publications:

- Jakob Henriksson and Florian Heidenreich and Steffen Zschaler and Jendrik Johannes and Uwe Aßmann. *Extending Grammars and Metamodels for Reuse – The Reuseware approach.* Special issue on Language Engineering in IET Software Journal, Volume 2, 2008.
- Jakob Henriksson and Jendrik Johannes and Steffen Zschaler and Uwe Aßmann. *Reuseware - Adding Modularity to Your Language of Choice.* TOOLS EUROPE 2007 - Objects, Models, Components, Patterns. Zurich, Switzerland, June 2007.
- Jakob Henriksson and Florian Heidenreich and Jendrik Johannes and Steffen Zschaler and Uwe Aßmann. *How dark should a component black box be? The Reuseware Answer.* Proceedings of the 12th International Workshop on Component-Oriented Programming (WCOP). Co-located with 21st European Conference on Object-Oriented Programming (ECOOP'07). Berlin, Germany, July 31 2007.
- Jakob Henriksson and Michael Pradel and Steffen Zschaler and Jeff Z. Pan. *Ontology Design and Reuse with Ontological Roles.* In Proceedings of the Second International Conference on Web Reasoning and Rule Systems (RR'08). Karlsruhe, Germany, October 2008.
- Uwe Aßmann and Sacha Berger and François Bry and Tim Furche and Jakob Henriksson and Jendrik Johannes. *Modular Web Queries—From Rules to Stores.* In Proceedings of On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, volume 4806/2007, pages 1165–1175, Lecture Notes in Computer Science.

Contents

List of Tables

List of Figures

Part I

Overview

There are substantial benefits to introducing a component approach even in cases where component markets or in-house component reuse is not yet foreseeable.

Clemens Szyperski
(Component Software, 2nd edition)

1 Introduction

In the early 1980s, Tim Berners-Lee developed the first system of inter-linked hypertext documents, today better known as the World Wide Web—the hugely successful network of information and services now part of our daily lives. With vast amounts of data available on the Web it is vital to employ computer systems and software to find, organize and display data for the benefit of the human user. The perhaps most visible such system today is the Web search engine, to many considered indispensable for finding information. An inherent drawback with any computerized system dealing with information is the system's lack of understanding of what the information actually means. Failing to understand the intended meaning of information often leads to the inability to properly process and present that information. On the Web, for example, software agents can only extract the syntactical structure of a Web page while a human quickly can capture the often subtle and deeper intended meaning of the content—the semantics. Web pages are traditionally written for humans by humans, unfortunately leaving a semantic gap between the human users and software systems employed to help better process the available content.

An initiative addressing this issue was published by Berners-Lee in an 1999 edition of Scientific American.[1] The article outlined a refined model for the Web where not only humans could read and understand the available information, but also machines. To signify the shift in importance from the information itself to the meaning and intention of the same, the term 'Semantic Web' was coined.

> "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."
>
> *– Tim Berners-Lee*

Since its conception, the Semantic Web has come to encompass many different ideas, research topics and technical solutions—all devoted to promoting semantics on

[1] `http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21`

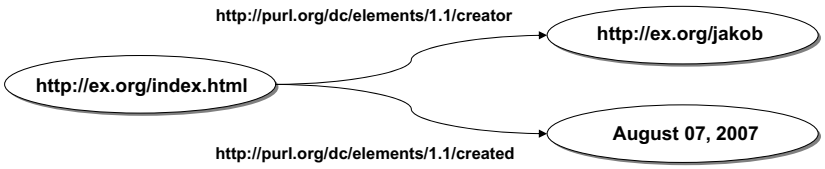


FIGURE 1.1: *A Web site given metadata about its creator and creation date.*

the Web.[2] The original core idea of the Semantic Web, however, remains the same: to provide well-defined *metadata* (data about data) to allow software agents to accurately process the underlying data. As such, it can be argued that the Semantic Web encompasses a fundamental duality in its realization—two sides of the same coin that will remain unchanged irrespective of specific adopted technical solutions: One part deals with formulating well-defined metadata, the other with accessing and processing that metadata for the benefit of the users. To specify and encode metadata a suitable descriptive language needs to be employed; to access and extract the metadata (or the underlying data itself) an appropriate query language be must available. Such languages are the fundamental tools—the chisel and hammer—of the Semantic Web.

In order for machines to make use of the provided metadata it needs to be expressed in a machine-processable format using a formal language. A formal language can provide an unambiguous description of the data and allow for reasoning. Thus, if this is the case, software agents can process the metadata, in some sense understand it and possibly draw useful conclusions based on the given information. The Resource Description Framework (RDF) [55] is an example of such a metadata language. RDF is designed to represent information about different kinds of resources identified by Uniform Resource Identifiers (URIs). RDF encodes information in simple triple statements consisting of a subject, a predicate and an object. An example of two such statements are shown in graphical form in Figure 1.1. The statements provide metadata about the Web site `http://ex.org/index.html`, stating that it was created on August 7, 2007 by the person identified by the resource `http://ex.org/jakob`.

Ontologies have emerged as a viable approach and formalism for modeling metadata. The currently adopted ontological formalisms are based on first-order logic (FOL) and rely on many years of research in the field of knowledge representation, particularly on the Description Logics (DLs) formalism [8]. Web-adapted versions of DLs with machine-processable serializations have also been standardized by the World Wide Web Consortium (W3C), for example, the Web Ontology Language OWL [73] which is (partly) layered on top of RDF. Ontologies are commonly processed by dedicated reasoning engines. Such dedicated engines are often powerful, but can also be too heavyweight for some applications, where speed and efficiency is of essence. The applied reasoners may be used for checking consistency of ontologies or for drawing new implicit conclusions from the explicitly given information. However, since the W3C-championed OWL language serializes its ontologies in a format based on the Extensible Markup Language (XML), they can also be processed by general-purpose XML query engines. Queries can be written that processes an ontology document and uses the extracted information in some intelligent way. As a simple example, if the metadata

[2]For an overview of current Semantic Web topics a "Semantic Web Topic Hierarchy" has been defined at `http://semanticweb.org/wiki/Semantic_Web_Topic_Hierarchy` (accessed 16 September 2008).

in Figure 1.1 was written in OWL, the OWL document could be queried by an XML query engine for the creator of the Web page `http://ex.org/index.html`, and would return `http://ex.org/jakob` as the answer. For data intensive tasks, general-purpose (XML) query languages can be more efficient and hence be preferable over dedicated reasoner. So, while dedicated reasoners are better suited for complex reasoning tasks, general-purpose query languages can still provide some basic form of reasoning, and might be easier to use since they are already often deployed for accessing Web data (not metadata).

Domain-specific languages Ontology and query languages can both be seen as examples of *domain-specific languages* (DSLs). DSLs are pervading many application areas and are often considered to be a great tool for helping to cope with complexity in software development. DSLs can help with respect to complexity by allowing to program, model, or specify certain software parts using succinct and domain-appropriate language constructs. A complex Web application might—as one part of its realization—make use of an appropriate XML query language for data access and transformation, for example, XQuery [13]. Another popular DSL is LINQ (Language Integrated Querying) which allow query programmers to write database queries in C# programs, but using an intuitive syntax very similar to SQL (Structured Query Language) queries [14]. A domain-specific language is defined in [28] as:

> "[A] programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular domain."
>
> *– van Deursen, Klint, Visser [28, p. 26]*

As the authors of [28] point out in their work, the key to the definition is the notion of *focused* expressive power. Thus, constructs and abstractions are provided specifically for formulating and solving problems related to the domain for which the language was developed. With such specialized constructs and appropriate domain-related abstractions at hand, programmers can concisely express what they want and need. Even though DSLs can be a tool to help cope with software complexity, unfortunately, they also introduce a new level of complexity that is not always initially foreseen. The new complexity arises because the DSL specifications themselves may grow in size. For example, XML query or transformation programs can easily grow large and become hard to manage and maintain. As those specific parts of the larger software applications grow, there must be means in place to cope with that growth in order for DSLs to maintain their attractiveness. Admittedly, some DSL specifications will never grow in this fashion, but as explained, there are some that will.

Domain-specific languages stand in direct contrast to *general-purpose languages* (GPLs), such as Java or UML. General-purpose languages differ from domain-specific ones in (at least) one interesting aspect: rich *abstractions*. First of all, general-purpose languages—at least programming-oriented GPLs such as Java—usually relate to some specific programming paradigm(s), they are declarative, imperative, procedural, object-oriented etc. Regardless of the paradigm choice, the languages do not beforehand assume the type of problems they will be used to model and address. Instead, within the context of their paradigms, they focus on more general constructs for structuring programs and ways of defining reusable units, for example: functions, methods, classes,

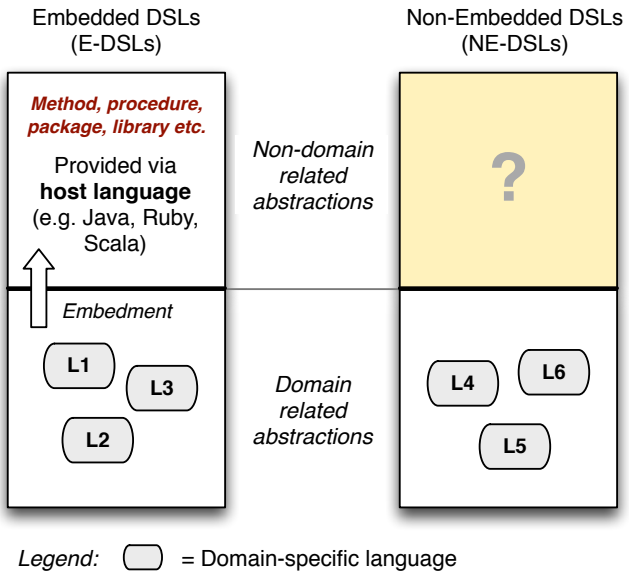


FIGURE 1.2: *A host language provides non–domain-specific abstraction constructs for an embedded DSL. This option is however not available for a non-embedded DSL.*

macros, templates, packages. The situation is however not the same for domain-specific languages, in fact, the opposite often holds. As domain-specific languages are developed to model specific kinds of problems—and have *focused* expressive power—they are not always pre-equipped with more general *non–domain-specific* abstraction and reuse constructs in the way many general-purpose languages are. There are at least two reasons for this. The first reason is that language and tool development is difficult, and foremost costly; the inclusion of any additional constructs is a hard balance between the actual need and incurred cost of introducing them. The second reason is that domain-specific languages are commonly developed with the intention of being embedded in some general-purpose *host language*. The language is then referred to as an *embedded* domain-specific language (E-DSL) [47]. One of the motivations for embedding a domain-specific language is to reduce its development cost. The advantage of embedment is that both syntax and semantics from the host language can be reused for the domain-specific language. For example, as a way to provide semantics for the embedded language, a translation into the host language can be specified. In that case, thanks to the translational semantics, existing tooling for the general-purpose language may be reused. Often an explicit translation is not needed at all. Instead the flexible syntax of the host language is simply used to accomplish desirable statement forms that suffice for the purpose of the DSL. Host languages that are suitable for this kind of development are for example Ruby and Scala.[3] As an added benefit, the already existing abstraction and reuse constructs of the host language may be exploited, making it unnecessary to provide them in the domain-specific language in the first place (see left-hand side of Figure 1.2). This also holds for other useful language constructs, such as control-flow mechanisms (e.g. conditionals and loops).

[3]See `http://www.ruby-lang.org/` and `http://www.scala-lang.org`, respectively.

However, there also exist domain-specific languages not intended to be embedded into a host language. This can be the case if there is no obvious host language candidate, or because the DSL is intended to be used as a standalone language. We call such a language a *non-embedded* domain-specific language (NE-DSL).[4] In this case, new tooling has to be developed at the normal high cost. Thus, most investments are usually centered on the core *focus* of the language. For a query language, for example, the central query algorithms are prioritized; for an ontology language the reasoning mechanisms receive the attention. Thus, rich non–domain-specific abstractions and reuse constructs are not always directly available and can be difficult to include in the language at low cost.

1.1 Problem: Component-based development for DSLs

This situation—a non-embedded domain-specific language without easily achievable reuse constructs—corresponds to the top-right quadrant of Figure 1.2 and is part of the problem we address in this thesis. We will in the course of this thesis give examples of two (Semantic) Web related languages that fit this description and that are in need of such rich abstractions and reuse constructs.

Component-based Development with DSLs In the above we have essentially already equalled the notions of abstraction and reusability. This is confirmed by Wegner who states that "abstraction and reusability are two sides of the same coin" ([93, p. 30] as cited in [60]). Krueger paraphrases Wegner explaining that "every abstraction describes a related collection of reusable entities and that every related collection of reusable entities determines an abstraction" [60]. From reusable entities the step to the general notion of "components" and component-oriented development is not far. Component-oriented development has played a major role in the traditional software engineering discipline [86]. Large software products benefit drastically in terms of robustness and maintainability if built from reusable, understandable and independently designed units.

This is not only true in traditional software engineering, but is equally valid on the Semantic Web. Ontologies can often be large descriptions that need to be maintained and understood, just like any other software. As an example, the well-known Gene Ontology [88] describes approximately 25,000 terms and their relationships (as of January 2008). Furthermore, if large ontologies are not constructed from smaller components, reusability of already modeled parts is hampered. The same holds for Web query languages. Queries in traditional database settings, for example expressed in SQL, do not usually depend on the data-size and can therefore remain small and do not always have a direct need for modularization and reuse. The orthogonality of query size and data size is also true on the Web, but with an important difference. Queries on the Web often need to respect many different, and not seldom exotic, data schemata. This means that query programs cannot always make assumptions of exactly how data is represented—according to which schema the data is modeled. Query developers wanting to develop robust query programs need to take this into consideration, often resulting in large and verbose query programs.

[4] Martin Fowler makes the distinction between *internal* and *external* DSLs, roughly corresponding to E-DSLs and NE-DSLs [32]. NE-DSLs are also sometimes called *standalone*.

An example of such a situation is the possibility to serialize the same OWL ontology in different ways.[5] The different ways OWL ontologies can be serialized should be respected by query programs working on the ontologies. Enabling the possibility to encapsulate part of a query program dealing with a particular data schema (serialization) becomes important. Encapsulation of this information allows for better maintainability, since modifications to query programs can be localized to independent components.

Components and component-oriented development is not only needed for ontology and query languages on the Web. It can also be important for domain-specific languages in general, but as already argued it is especially pressing for languages not intended to be embedded in a general-purpose host language (see Figure 1.2). The number of such non-embedded languages developed for the Web is high, most likely because there is no obvious and appropriate host language into which to embed them. But it should be noted that such languages also exist in other areas, such as software modeling [39]. As a remark, many abstractions and reuse constructs that could be useful for programmers are not even available in many general-purpose languages, either because the designers did not want to integrate them, or because it is costly and difficult to do so. Examples of such constructs are roles [83], aspects [51, 52], mixin layers [80] and traits [79]. As mentioned, the development of domain-specific languages often have a different focus than general-purpose languages; the focus is on the essential primitives needed for modeling problems related to the domain in question. More general constructs are not usually, at least initially, taken into consideration. Such constructs are often not really essential to programmers at the early stages when the language takes its first staggering steps towards being truly productive. Nonetheless, when a domain-specific language has been developed and is being used by programmers to build larger software artifacts, needs for abstraction and reuse become inevitable.

1.2 Thesis Contributions

The work presented in this thesis is a response to one of the challenges put forward in the European Network of Excellence REWERSE[6]: develop composition technology for query and ontology languages used on the Semantic Web. In addressing these issues we have developed composition technology applicable to a wide range of languages. But, in demonstrating the technology we have focused on two languages in particular. For querying we have focused on the language Xcerpt [77], the further development of which was another main objective of REWERSE. Xcerpt is also an interesting choice because of its declarative and rule-based nature (rules are considered an important paradigm on the Semantic Web), and because of its lack of an explicit control flow mechanism. For ontologies we have focused on the Web Ontology Language OWL [73]. This choice is quite natural since OWL is the main ontology language being used today and is standardized by W3C.

In the following we give a brief summary of what will be described in more detail in subsequent chapters. We here try to focus on the conceptual advance presented in this thesis.

[5] By 'serialize' we here mean the encoding of the ontology in an XML format.

[6] Reasoning on the Web with Rules and Semantics, 2004–2008. `http://rewerse.net`.

1.2.1 Composition Technology

Our developed composition technology builds upon previous work. BETA is an object-oriented programming language which first proposed the idea of *grammar-based modularization* (GBM) [63]. The technique is called grammar-based because the considered modules—pieces of source code, called *fragments*—are defined wrt. an underlying language grammar. Such fragments only have *explicit interfaces*. We will not just yet detail what these interfaces are and how they can be used, other than to say that the interfaces—ways of integrating fragments into larger programs—are always declared by their authors. To be precise, the grammar-based modularization capabilities are not integrated into the BETA language itself, but is rather part of its development environment: the Mjølner system. So, the Mjølner system realizes the said modularization technique for the BETA language. But in fact the technique is very general and, in principle, not restricted to only BETA. However, the Mjølner system does not provide a way for working with arbitrary languages. We develop a generative approach for extending language grammars such that the GBM techniques can be applied. We assume that the grammars are context-free. It is generative in the sense that based on some user input relating to the desired modularization possibilities, an extended grammar is generated which describes a language in which it is possible to define the required fragments. We call the technique *universal grammar-based modularization* (U-GBM). The technique is referred to as being 'universal' since arbitrary languages can be addressed. We will always use the term 'universal,' or the verb 'universalize,' in this sense. So, while the Mjølner system is *grammar-based* and general in its conceptualization, we make the development of GBM-based systems *grammar-driven* and general in practice.

Having this possibility to enable modularization—separation of software artifacts into "components"—for arbitrary languages is important. Especially when component-based development has not initially been planned for. This is also acknowledged by Clemens Szyperski. Even though Szyperski's notion of components differs from that of GBM, his claim should still be valid [86]:

> "There are substantial benefits to introducing a component approach even in cases where component markets or in-house component reuse is not yet foreseeable."
>
> *– Clemens Szyperski, Component Software [86, p. 139]*

Szyperski essentially says that a component approach is needed sooner or later. With universal GBM, a component approach can be introduced into arbitrary languages. Not necessarily at the time of language design, but even afterwards.

Definition of components (hence a form of modularization) is supported in *invasive software composition* (ISC) [5] by a technique that is similar to the one in GBM, but where *implicit interfaces* are also considered. As in GBM, components in ISC are fragments. Implicit interfaces are points in fragments that are not declared by the fragments' authors, but which still are accessible during the composition process. For this reason, ISC can simulate techniques such as aspect-oriented programming [51], which is traditionally completely reliant on the notion of implicit interfaces. As such, ISC is a more flexible and powerful approach than GBM. We leverage our 'universal' extension of GBM to also cover ISC. First of all, this makes the development of ISC-based systems *grammar-driven*. As a consequence, we achieve *universal invasive software composition* (U-ISC). That is, we enable the possibility of applying ISC to arbitrary

languages. Being able to generalize the technique to arbitrary languages in a grammar-driven manner does not only make it easier to build ISC-based composition systems for new languages, but it also improves our understanding of the underlying techniques and how well they work for arbitrary languages.

One of the interesting results from ISC is its distillation of two basic composition operators that are used to assemble fragments into useful programs. These operators are very general, that is, language independent, which is a basic requirement for a 'universal' approach. However, Aßmann explains:

> "[T]he basic operators are not expressive enough, since they are so general. [...] Software designers will not like designing with a minimal pattern language. Instead they will need languages with more domain-specific, tailored, and adequate composition operators. [...] And I believe that such languages will be the software construction languages of the future."
>
> *– Uwe Aßmann, Invasive Software Composition [5, p. 278]*

One of the root causes for this problem is that source code fragments, the module type of choice for both GBM and ISC, are inappropriate abstractions for most end-users and programmers. But the composition algebras of both these approaches can only work on the primitive level of transforming fragments, using the "minimal pattern language." To address this problem we develop a technique which allow programmers to use more appropriate abstractions; abstractions that are related to the programming languages they use on a daily basis (for example, their DSLs). Such abstractions are specified in *extensions* of programmers' languages, extensions tailored for the kind of component-based development they are in need of. Included in such extensions are the "domain-specific, tailored, and adequate composition operators." But, unbeknownst to users of such an extended language, the extended programs that support the appropriate modularization constructs are composed into semantically equivalent programs of the underlying, non-extended, language. So, the programs written in an extended language are given a reduction semantics by referring to the underlying language. We call this approach *embedded invasive software composition* (E-ISC). The reason we call it 'embedded' is because the reduction semantics is defined by a U-ISC–based composition system. So, we exploit and build upon the previous work on U-GBM and U-ISC to reach this higher goal.

Hence, the composition technology contribution of this thesis is a three-staged advance from previous work. This advance is illustrated in Figure 1.3. First, we universalize GBM to achieve universal GBM (U-GBM). Then, we build upon this to make ISC grammar-based, and, in the same sense, universal (U-ISC). Finally, due to the difficultness of working with fragments as first-class software artifacts, we define embedded ISC which allow end-users to work with more intuitive software units (E-ISC). This lays the foundation for addressing an important open issue, that is, how to enable component-based development for NE-DSLs.

One immediate consequence of the above-mentioned reduction semantics is that the core expressiveness of the addressed language is never really extended. One great benefit of this is that existing tooling can be reused. We do not claim that the expressiveness of DSLs is at fault, rather their ability to support developers in defining and using reusable entities—components. This is supported by another remark from

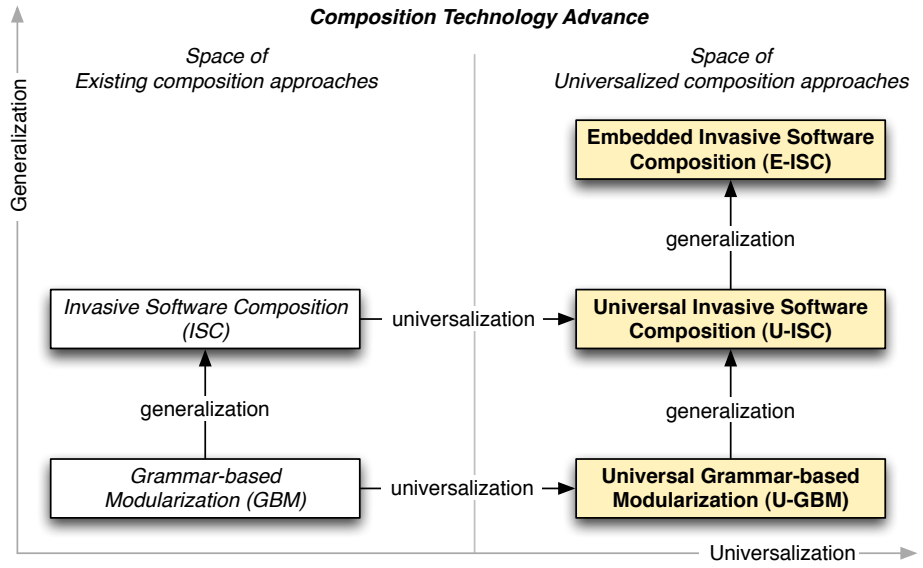


FIGURE 1.3: *We present a three-staged advance over previous work:* universal grammar-based modularization, universal invasive software composition and embedded invasive software composition.

Szyperski:

> "[...] from a purely formal point of view, there is nothing that could be done with components that could not be done without them."
>
> *– Clemens Szyperski, Component Software [86, p. 10]*

The universal nature of our composition approach essentially means that we are defining a composition *framework*. The following are some benefits with our framework approach towards enabling component-oriented development and design for programming language in general, and domain-specific languages in particular:

- By providing a framework approach where the exact properties of the addressed language are not presumed, it is possible to augment any language that has a context-free grammar with new abstractions. Not only already existing and well-known abstractions, but also new abstractions which will be found to be useful for existing or future domain-specific languages.
- A consequence of a strictly reductional semantics for extended language construct is that the approach allows for the possibility of introducing new abstractions for languages where associated tooling (compilers, interpreters etc.) are not available to be modified, as for example in proprietary systems. Here, new requirements for abstractions may arise for users of the language/system, but the organization/company holding rights to the tools are not planning to introduce the requirements. Our approach gives a solution to such problems by allowing independent augmentation of abstraction and reuse of a language, without having to modify the associated tooling.

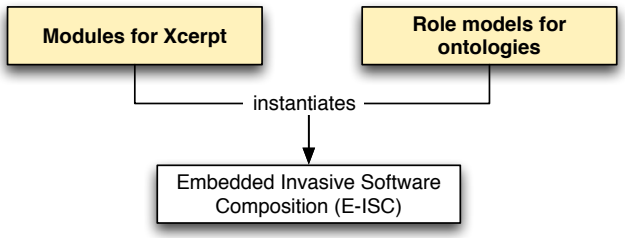


FIGURE 1.4: *To evaluate our composition framework we instantiate it twice for two different languages and abstraction concepts:* modules *for the query language Xcerpt, and* role models *for ontology languages.*

– There might still be cases where developers of a language and its associated tools decide that a newly designed abstraction should be tightly integrated into the language and tools. For example, due to strict requirements on efficiency. As such an integration incurs costs in terms of time and money, it might be beneficial to *prototype* the concepts before actually performing the full integration. This prototyping could be done using our framework.

We demonstrate the applicability of our framework by instantiating it twice, for two completely different languages and abstraction concepts (see Figure 1.4 for an illustration). These framework applications hence evaluate the framework, and are briefly mentioned below.

1.2.2 Evaluation 1: Modules for Xcerpt

We demonstrate the usefulness of allowing Web query developers to program with *modules*, an arguably necessity on the Web. The particular language we address is the rule-based Web query and transformation language Xcerpt [77]. Modules are sets of related rules and can be defined using intuitive syntax. Moreover, since modules should be encapsulated to ensure proper separation of concern, module interfaces can also be defined. Apart from allowing to define reusable query components in form of modules, constructs are provided to import and deploy them. We give several examples of the usage of the introduced constructs, as well as explain their semantics. In practical terms the Xcerpt module system is realized through our composition technology.

1.2.3 Evaluation 2: Role Models for Ontologies

We introduce and discuss a new reuse unit for ontologies—*role models*. A role model is an abstraction unit that has previously been investigated in data, object-oriented and conceptual modeling (see, e.g. [83] and references therein). However, role modeling is lacking in current ontology languages, such as OWL [73]. We introduce the notion of role modeling into ontology languages and show how it can be used to modularize ontologies. We demonstrate the use of role models on a subset of OWL, but the approach is general and not limited to a particular ontology language. We demonstrate a composition system developed using our composition technology that is able to compose ontologies from role models.

1.3 Thesis scope

In the following we comment on what is covered in this thesis, and what is not:

- *Language engineering.* Despite the fact that this thesis is about languages, and extensions of languages, it is not a thesis on *language engineering*. There are many sub-disciplines of language engineering that are employed in this work, but most of them fail to receive any serious attention, for example, language development, parsing theory, parser generation and grammar composition. The only highlighted language engineering concept is context-free grammars (CFGs). This is the underlying formalism any considered language is assumed to be expressible in. (Concretely this would be done in EBNF [1] or a similar practical formalism.) Language extensions for adapting languages to the composition techniques of, first grammar-based modularization, and then invasive software composition, are described by referring to CFGs. For this reason, CFGs receive certain attention in Chapter 2.

- *Ensuring valid compositions.* When composing software, it is important to guarantee certain properties of the final result. We will guarantee that the composition result is a syntactically valid program wrt. some underlying language. We will not, however, guarantee that the composition result is also semantically valid. For example, we do not, during the composition process, check the semantics of the composition results. The reason we do not provide this second level of guarantee is because we are developing a general framework addressing arbitrary languages. Since semantics is very specific for each particular language, such details would have to be provided for each language, in addition to the grammar specifications. This level of language tailoring is currently not provided in our general solution and is out of the scope of this thesis.

- *Composition framework evaluation.* While it is our belief that the presented composition framework can be instantiated for many different languages, it is here mainly validated for two languages: the Web query language Xcerpt and the ontology language OWL. However, it should be noted that the languages—apart from belonging to the set of languages we consider to be in need of modularization techniques—are quite different from each other in terms of their structure and purpose. Also, because they are completely different languages, the developed types of components for the languages are of entirely different natures. These differences wrt. languages and component types for which the framework is validated should be regarded in its favor.

Part II

Composition Framework

2

Universal Grammar-Based Modularization

BETA is an object-oriented programming language, historically following the SIMULA tradition in the Scandinavian school of object-orientation [57, 63]. One of the interesting concepts of BETA is the support of a single abstraction construct: the *pattern*. However, rather than delving into the details of BETA as an object-oriented language, we shall focus on another aspect of it, namely its program text modularization technique. One of the side projects developed around BETA was the Mjølner BETA system. The Mjølner system provides a *fragment system* (or *fragment language*) aimed for modularization of BETA program text. Essentially, any snippet of BETA source code—a *fragment*—can be a module. By putting such fragments together, a complete and executable program can be constructed. By separating the BETA language with its pattern construct from the modularization of program texts as provided by the fragment language, a separation between "programming in the small" and "programming in the large" is achieved [27].

The technique used by the Mjølner fragment system is called *grammar-based modularization* (GBM) in the literature [58, 59, 63]. The technique was initially introduced under the name *syntax-directed modularization* [59]. The technique is 'grammar-based' since the underlying language grammar dictates what are considered valid and deployable fragments for the modularization process. Some of the mentioned benefits for introducing this modularization technique to BETA were [63, Chapter 17]:

- Large programs are easier to understand, and edit, if split into a number of smaller, logically coherent, units. Development is also simplified if several people are working on the same project.
- Modules can be saved in a library and shared by several programs. Good modularization mechanisms will thus improve *reusability* of code as well as designs.
- It is good practice to split a module into *interface modules* and *implementation modules*. An interface module defines how a module can be used, and an im-

plementation module describes how a module is implemented. This makes it possible to prevent users of a module from seeing details about data representation and implementation of algorithms.

Despite of BETA having been introduced in the early 80s, we are not aware of any other language or system supporting the same particular technique for program modularization (except for *gbeta*, which is a direct extension of the BETA language and developed within the same group [29]). The fact that BETA itself is considered a 'dead' language does not help championing the particular modularization technique it was first, and practically alone, to provide.

The fragment system provided in the Mjølner system is directly connected to the BETA language itself, in particular to its grammar – fragments defined must conform to the BETA grammar. Nevertheless, the modularization approach, being based on the notion of grammars, is very general and in principle not restricted to the BETA language. This is indeed also mentioned by the developers of BETA [63, p. 256], but a concrete, practical and automatic technique for enabling the approach for an arbitrary language has, to the best of our knowledge, never been realized. Still, the generality of the modularization approach is intriguing. So, one interesting research question is how the grammar-based modularization technique of the Mjølner fragment system can be made available to arbitrary languages, not only in theory, but also in practice. That is, a method and a solution to the following scenario:

> Given a grammar G that specifies a language L, and a set of constructs in L that are considered useful modularization units, produce a grammar $G+$, derived from G, that specifies a language in which it is possible to write fragments (wrt. G) such that programs of L can be modularized.

The Mjølner fragment system provides a tailored solution to the above scenario, addressing the BETA language, its grammar, and certain constructs of the language. In this chapter we generalize this solution. The main contributions of this chapter are:

1. We provide an understanding of how grammar-based modularization, in the style of the Mjølner fragment system, can be made available to arbitrary programming languages.
2. We provide a practical transformation technique for adapting grammars such that it is possible to concretely formulate, and thus program with, the considered fragments.
3. We discuss how the original grammar can be used to ensure safe assemblage of fragments. We also discuss how programmers can be more precise in controlling what is considered safe for their fragments.
4. We analyze the minimal requirements for a fragment language. That is, what is required from a language used to specify how fragments are put together into meaningful programs.

By addressing arbitrary programming languages, we can achieve a *universal grammar-based modularization* approach (U-GBM). The goal is in particular to have a practical approach where specified grammars can be transformed, or adapted, in a general way to take advantage of GBM. The transformed grammars can be said to encode fragment systems, and realize the possibility of formulating concrete fragments. In this

sense we develop a *framework* for how the GBM approach can be realized for arbitrary languages. We use the term 'framework' here in the sense of capturing a *methodology*, rather than a particular design.

We will focus on languages that (syntactically) can be described by context-free grammars. Most programming languages can be described by such grammars. The investigations in this chapter are important because we will build upon them in subsequent chapters to provide more advanced and useful modularization and composition methods.

This chapter is structured as follows. In Section 2.1 we provide required background knowledge relating to context-free grammars and GBM. In Section 2.2 we then describe how context-free grammars can be adapted such that they encode fragment systems. In Section 2.3 we discuss how fragment systems can guarantee safe modularization. Finally, in Section 2.4, we summarize the achievements of the chapter.

2.1 Background

First we will recall the formalism of context-free grammars, since we will base our framework on it. Then, in Section 2.1.2, we look in more detail at the BETA Mjølner system and how its fragment system works.

2.1.1 Context-free grammars and languages

The syntax of formal languages used in computer science, for example programming languages such as Java, are often specified in some grammar formalism. A grammar is a convenient way of specifying an infinite set of possible programs in a finite way. This essentially means that there is not an a priori set of programs that can be written in the language being specified. The *Extended Backhus-Naur Form* (EBNF) [1] is a format frequently used to specify grammars. For example, the syntax of a simple fictitious rule-based language *RL*—*RL* for "Rule Language"—can be specified by the EBNF statements in Example 2.1.[1]

Example 2.1. (RL *grammar*) The following is an EBNF grammar for an example rule language.

⟨*prgm*⟩ ::= ⟨*stmt*⟩*

⟨*stmt*⟩ ::= ⟨*rule*⟩ | ⟨*fact*⟩

⟨*rule*⟩ ::= ⟨*head*⟩ `:-` ⟨*body*⟩ `.`

⟨*head*⟩ ::= ⟨*atom*⟩

⟨*fact*⟩ ::= ⟨*atom*⟩ `.`

⟨*body*⟩ ::= ⟨*atom*⟩ (`,` ⟨*atom*⟩)*

⟨*atom*⟩ ::= ⟨*predname*⟩ `(` ⟨*term*⟩ (`,` ⟨*term*⟩)* `)`

⟨*term*⟩ ::= ⟨*const*⟩ | ⟨*var*⟩ | ⟨*num*⟩

⟨*predname*⟩ ::= STRING

⟨*const*⟩ ::= STRING

⟨*var*⟩ ::= CAP_STRING

[1] In fact, our example language *RL* is very similar to the language Datalog.

⟨*prgm*⟩
⟨*stmt*⟩
⟨*rule*⟩
⟨*head*⟩ `:-` ⟨*body*⟩ `.`
⟨*atom*⟩ `:-` ⟨*body*⟩ `.`
⟨*predname*⟩`(`⟨*term*⟩`)` `:-` ⟨*body*⟩ `.`
`animal(`⟨*term*⟩`)` `:-` ⟨*body*⟩ `.`
`animal(`⟨*var*⟩`)` `:-` ⟨*body*⟩ `.`
`animal(X)` `:-` ⟨*body*⟩ `.`
`animal(X)` `:-` ⟨*atom*⟩ `.`
`animal(X)` `:-` ⟨*predname*⟩`(`⟨*term*⟩`)` `.`
`animal(X)` `:-` `tiger(`⟨*term*⟩`)` `.`
`animal(X)` `:-` `tiger(`⟨*var*⟩`)` `.`
`animal(X)` `:-` `tiger(X)` `.`

TABLE 2.1: *Derivation sequence from nonterminal* prgm *to the string in* (2.1).

⟨*num*⟩ ::= NUM_STRING

The first grammar rule states that a *program* (represented by ⟨*prgm*⟩) consists of an arbitrary number of statements (⟨*stmt*⟩). The second grammar rule states that a *statement* is either a *rule statement* or a *fact statement*. The third rule states that a rule statement consists of a *head*, followed by the word "`:-`", followed by a *body*, and trailed by a dot ("`.`"), and so on. In EBNF, an asterisk ($*$) after a nonterminal ⟨*n*⟩ usually mean "zero or more" of ⟨*n*⟩. A plus sign ($+$) means "at least one", and a questions mark (?) "zero or one." Hence, an *RL* program consists of "zero or more" statements. The traditional way of understand rules (here represented by ⟨*rule*⟩) is: If the body holds, then the head also holds. Facts are always true. The last four grammar rules define what predicate names, constant symbols, variables and numbers look like. They are defined by special tokens not further specified here, only to say that predicate names and constant symbols are character strings assumed to start with a lower-case letter, variables are capitalized character strings (first character must be upper-case), and numbers are strings of numerals.

■

We shall use Example 2.1 as an example grammar throughout this chapter. The ⟨*xyz*⟩ parts of the specification are called *nonterminal* symbols. Each nonterminal ⟨*xyz*⟩ generates a set of strings over a finite alphabet Σ, using the grammar rules with ⟨*xyz*⟩ on the left-hand side as a starting point. Rule *choices* (separated by `|`) describe different ways of generating the strings.

```
animal(X) :- tiger(X).
```
(2.1)

For example, the string in (2.1)—stating that every tiger is an animal—can be generated from the nonterminal ⟨*prgm*⟩ using the grammar rules. This is shown by the derivation sequence in Table 2.1.1, starting with the nonterminal ⟨*prgm*⟩ and ending with the string itself.

Each step in the sequence is derived from the previous one by replacing one of the nonterminals with the right-hand side of its definition.[2] Other strings can be generated

[2]This technique of matching a string with a grammar is also called *top-down parsing* in the literature [3].

in a similar fashion. The set of strings that can be derived from a carefully selected nonterminal of a grammar—called the *start symbol*—is called the *language* generated by the grammar.

Formally, a context-free grammar (CFG) is a 4-tuple [56]:

$$G = (N, \Sigma, P, S)$$

where N a finite set of nonterminal symbols, sometimes called *syntactic categories*, Σ is a finite set of terminal symbols (disjoint from N), P a finite set of production rules $N \times (N \cup \Sigma)^*$ and $S \in N$ the start symbol. Each production rule $N \times (N \cup \Sigma)^*$ can be used to rewrite N by $(N \cup \Sigma)^*$ (cf. example in Table 2.1.1). Any string in $(N \cup \Sigma)^*$ derivable from the start symbol S is called a *sentential form*. A sentential form that does not contain any nonterminal symbols is called a *sentence* (it only contains terminal symbols, that is, it is in Σ^*). For example, all the steps in the derivation sequence in Table 2.1.1 are sentential forms of the grammar in Example 2.1. Only the last step is a sentence of the same grammar (since it does not contain any nonterminal symbols). The start symbol S specified by a grammar G is of importance since it explicitly defines the valid *units* of the language generated by G, hence the valid units that can be specified by programmers. Sentences derived from S are sometimes called *programs* of the language. In Example 2.1 the nonterminal *prgm* is assumed to be the start symbol.

Considering a CFG $G = (N, \Sigma, P, S)$, a sentential form F_1 derives a sentential form F_2 in one derivation step if and only if $F_1 = \sigma_l A \sigma_r$, $F_2 = \sigma_l A_1 \cdots A_n \sigma_r$ and $A ::= A_1 \cdots A_n \in P$, where each σ_i is some sequence of terminal symbols and A_j are nonterminals. This is written as:

$$F_1 \xrightarrow[G]{1} F_2$$

where the one (1) above the derivation arrow signifies that the sentential form F_2 is derived from F_1 in one step. If the sentential form F_1 derives F_2 in any number of derivation steps, then this is written:

$$F_1 \xrightarrow[G]{*} F_2$$

The set of strings generated by a syntactic category n of a grammar G is called the *language* $L_G(n)$ of n (simply $L(n)$ when it is clear which G is meant). This can be defined as:

$$L_G(n) = \{x \in \Sigma^* \mid n \xrightarrow[G]{*} x\}$$

The *de facto* semantics of a context-free grammar is given by the sentences it generates [4]. The set of all sentences of a grammar G is called the *language* $L(G)$ *generated by* G and is defined as:

$$L(G) = \{x \in \Sigma^* \mid S \xrightarrow[G]{*} x\}$$

Notice that $L(G) \equiv L(S)$ where S is the start symbol of G. A language L is context-free if there exists a context-free grammar that generates it. Intuitively, a context-free grammar G (of a programming language L) defines a (possibly infinite) set of sentences (programs) that conform to G. Most programming languages can syntactically be defined by context-free grammars, and in the following we only consider such languages.

2.1.2 Grammar-Based Modularization (GBM)

The Mjølner system, for the purpose of the BETA programming language, proposed a grammar-based modularization technique [63]. The Mjølner *fragment system*, or *fragment language*, allows for the definition of BETA source code fragments as modules. To simplify the presentation we do not introduce the BETA language and its syntax, but instead exemplify the technique using our rule language *RL* defined in Example 2.1. We try to stay as close as possible to the terminology used in [63]. Modules, which here are equalled to fragments, are syntactical structures of the considered language and are called *forms*. Forms must belong to some syntactic category of the underlying grammar, and hence be derivable from some of its nonterminals. A form derived from nonterminal $\langle A \rangle$ is called an *A-form*. Forms can in principle be any sequence of terminal and nonterminal symbols of the considered grammar. Hence, forms are essentially sentential forms of a particular syntactic category of the grammar. The sentential form in (2.2) can be seen as a *rule*-form of the *RL* grammar with one $\langle num \rangle$ and one $\langle atom \rangle$ nonterminal (not yet derived to terminal symbols).

$$\texttt{bonus(X, } \langle num \rangle \texttt{) :- employee(X), } \langle atom \rangle \texttt{.} \tag{2.2}$$

To be able to refer to nonterminals in forms, they are given names. Nonterminals meant to be replaced by the fragment system are called *slots* and have the following syntax:[3]

$$\texttt{«SLOT T:}A\texttt{»} \tag{2.3}$$

where `T` is the name of the slot and A is its syntactic category. The sentential form from (2.2) can thus be written as in (2.4), which contains a slot named `value` of syntactic category $\langle num \rangle$ and a slot named `condition` of syntactic category $\langle atom \rangle$ (when using nonterminals in slots we do away with the angle brackets). These slots describe where change can take place and are called *slot declarations*.

$$\begin{aligned} &\texttt{bonus(X, «SLOT value:}num\texttt{») :- employee(X),} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\texttt{«SLOT condition:}atom\texttt{» .} \end{aligned} \tag{2.4}$$

When defining forms in the fragment system, they must be given a name and a syntactic category, and are then called *fragment-forms*. Following the style of [63], we use a graphical syntax for defining fragment-forms. The table in (2.5) demonstrates the graphical syntax (grayed table rows indicate 'meta' information about forms, while white rows contain concrete forms).[4]

F:A
ff

(2.5)

In (2.5) `F` is the name of the fragment-form, A is its syntactic category and `ff` is the form (derivable from nonterminal $\langle A \rangle$). The Mjølner system also introduces the notion of *fragment groups*, which are sets of fragment-forms associated by a name using the

[3] This syntax was originally chosen for its suitability wrt. the BETA language, and we use the same here.

[4] There is also a textual syntax available in the Mjølner BETA system, but is not further discussed here.

`name` construct (illustrated below). A fragment group containing a single fragment-form, corresponding to (2.4), is shown in (2.6).

name 'RuleGroup'
`myRule:`*`rule`*
`bonus(X, «SLOT value:`*`num`*`») :- employee(X),` `«SLOT condition:`*`atom`*`» .`

(2.6)

Complete programs are assembled by binding fragment-forms to declared slots. The *origin* construct can be used for this purpose. The *origin* construct takes the fragment group being operated on as an argument. The fragment-forms appearing in a fragment group with an *origin* construct are called *slot applications*.

name 'Rules'
origin 'RuleGroup'
`value:`*`num`*
`200`
`condition:`*`atom`*
`efficient(X)`

(2.7)

By matching the names of the fragment-forms in (2.7) (slot applications) with the slot names in the fragment group indicated by the *origin* construct (slot declarations), the fragment-form in (2.8) is constructed.

`myRule:`*`rule`*
`bonus(X, 200) :- employee(X), efficient(X).`

(2.8)

Notice that the form in (2.8) is a valid sentence of the *RL* language, stating that "efficient employees receive a bonus of 200." As such it is a useful entity constructed from its smaller fragment parts. The above has demonstrated the main idea of the Mjølner fragment system, but using the simple *RL* language rather than BETA itself. All the features of the fragment system have not been discussed here, instead we direct the reader to [63, Chapter 17] for further details.

The grammar-based modularization provided by the Mjølner fragment system is attractive because of its simplicity. All the variable points in fragment-forms are explicit via slot declarations, and hence clearly dictate where forms can be modified (but not how). A fragment-form with one slot allows for the variability dictated by the slot declaration. That is, any form of the same syntactic category as specified in the slot declaration can replace it. The technique also allow fragment-forms to separate between module *interface* and module *implementation*. The slot declarations are the interfaces, while the slot applications specify the implementations. Thus, the implementations can easily be varied by changing the slot applications. The main benefits of grammar-based modularization are thus:

– The underlying idea is simple and easy to understand for programmers.

– By being based on the notion of grammars the approach is very generic and can be applied to most programming languages.

– Since fragments only can be transformed via their declared slots, a kind of encapsulation is supported.

– Slots allow for rich variability by not restricting the content of the fragments being bound. This allows for separation between fragment interfaces (slots) and their realizations (slot applications).

– Assembly of fragments is safe (controlled by syntactic categories in a declarative way).

While the underlying idea is language independent, the Mjølner fragment system has its limitations due to its close connection to the BETA language:

– Slots can only be declared for a few carefully selected syntactic categories of the BETA grammar.

This restriction is in place because the fragment system supports separate compilation of fragment-forms. Hence, fragment-forms containing slots can first be compiled and then later bound together in the compiled form. The benefit is that the target fragment does not have to be recompiled if a slot application is changed. Any limitations to the general approach are in place in the Mjølner fragment system to serve particular needs of the BETA language and its users. But it should be noted that supporting, for example, separate compilation of fragments can have far-reaching consequences. For example, special tools have to be implemented to handle the grammar-based modularization technique that is special to the Mjølner fragment system.

2.2 Universal Grammar-Based Modularization

Our goal is to deploy the same grammar-based modularization technique as outlined in the previous section, but without the limitations connected to the Mjølner fragment system. In contrast, we pursue a lightweight approach. That is, we want to provide a methodology and technique for developing and generating *lightweight fragment systems*. Given a grammar and certain input regarding what kind of modularization should be supported (for example which fragment-forms should be definable), a grammar-specific fragment system can be generated allowing modularization of program text in the style of GBM. The approach is considered 'lightweight' because:

– Separate compilation of fragment-forms is not supported. While less powerful, the benefit is that existing tooling does not have to be extended to handle compilation of fragment-forms and slots appearing in such forms.

A consequence is that the lightweight approach is strictly static, in the sense that it can be seen as a front-end – fragments must be assembled into valid programs of the underlying language before being compiled/interpreted. Since the fragment system is not involved when assembled programs are executed, existing tools (e.g. compilers and interpreters) can directly be reused.

We expect the following benefits of a lightweight approach:

– Support for different languages does not in detail have to be programmed for each language's grammar. Instead, support for grammar-based modularization and programming with the slot concept can be given semi-automatically by appropriate transformations on the considered grammar, resulting in a grammar-specific fragment system.

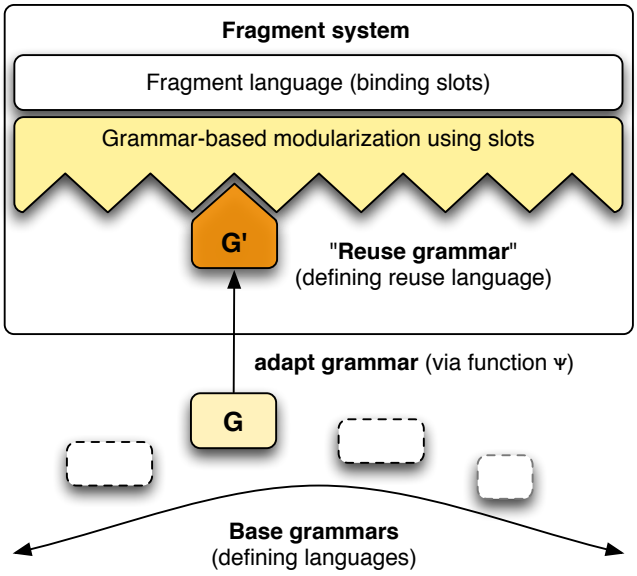


FIGURE 2.1: *Languages formally specified by a* base grammar *can be adapted to a* reuse grammar, *which generates a language that allows slots. Such a language is called a* reuse language *wrt. its* base language.

– In line with the conceptual idea of grammar-based modularization, slot support can be given for any syntactic category of the considered grammar, and not for only a few selected ones as in the Mjølner system.

– Any context-free language can be supported. Thanks to a generative approach, new languages can more quickly be addressed and supported with fragment systems. This will enable us to learn more about the true usefulness of the approach, since we can do 'rapid prototyping' and more easily experiment with different underlying languages.

In the following section we will discuss how this lightweight support for grammar-based modularization can be made available for arbitrary languages.

2.2.1 Grammar adaptation for GBM

In Section 2.1.1 we briefly introduced context-free grammars as the standard formalism for describing the syntax of formal languages. We also mentioned EBNF as a practical language for specifying such grammars. This section describes how formally specified languages can be adapted, or extended, to be subject to grammar-based modularization. This idea is illustrated in Figure 2.1 were `G` is the *base grammar* (specifying a *base language*) under consideration and `G'` its adaptation, called a *"reuse" grammar* (wrt. `G`), which generates a language in which it is possible to program with slots. Along with a fragment language (discussed in Section 2.2.2), such an adapted grammar essentially constitutes a lightweight *fragment system* supporting grammar-based modularization (see Figure 2.1). This is so because the reuse grammar describes what

valid fragments are and what their interfaces can be, while the fragment language can be used to describe how they should be assembled.

We start by recalling, and formalizing, what are considered to be valid and well-formed *units* of deployment in such an approach—fragments. First, we may define units that are not valid programs of the original language, but "subsets" thereof. For example, we might want to use *atom*s as basic building blocks (units) for a fragment system based on *RL*. So, while not valid programs wrt. their grammar, each such unit is a string derivable from some nonterminal of the grammar. For *RL atom*s, such a nonterminal would be $\langle atom \rangle$. For a given grammar G, we will call such a unit a P_n^G*-program* (or simply *P-program* when referring more abstractly to such a unit).

Definition 2.1. *(P_n^G-*program*) Given a CFG G, the strings $P \subseteq \Sigma^*$ derivable from nonterminal $n \in N$ of G are called P_n^G-programs.*

Notice that P_n^G-programs, in formal language theory terms, are nothing other than strings belonging to the language generated by the nonterminal n of grammar G, that is, strings in $L_G(n)$. Keep in mind that P-programs are sentential forms that are also sentences. The string "`efficient(X)`" is an example of a P_{atom}^{RL}-program.

Second, to allow for variability in fragments we may define P-programs that are incomplete "within" themselves via the concept of slots. Such slots effectively determine the possible variability for the P-programs. Formally, the slots are non-derived nonterminals.

Definition 2.2. *(F_n^G-*program*) Given a CFG G, the strings $F \subseteq (N \cup \Sigma)^*$ derivable from nonterminal $n \in N$ of G are called F_n^G-programs.*

That is, F_n^G-programs as defined in Definition 2.2 are *sentential forms* of G, but derived from $n \in N$ rather than the start symbol of G. The string "`p(`$\langle$`var`$\rangle$`) :- q(X),` $\langle$`atom`$\rangle$`.`" is an example of a F_{rule}^{RL}-program. It is not a P-program since it contains nonterminals.

Whenever it is clear what is meant we shall use the term *fragment* in a rather liberal fashion to mean P- or F-programs. That is, partial and possibly under-specified (via slots representing nonterminals) programs of some language (as defined in Definitions 2.1 and 2.2).

We notice that regardless of a specific grammar (or, programming language specified by the grammar), P-programs and F-programs are the first-class entities in any grammar-based modularization system or environment. That is, they are the fragments being composed and transformed to construct software.[5]

Grammar-based adaptation requirements In the following we briefly summarize the requirements on a lightweight adaptation of a base grammar to support grammar-based modularization.

1. *P- and F-program specification.* It must be possible for programmers to concretely define partial programs, that is P- and F-programs. This makes it possible to reuse and compose fragments, as defined in some fragment language (by a developer).

   (a) *P-programs.* It should be possible to restrict which kind of P-programs are allowed to be defined and worked with. That is, for which $n \in N$ wrt.

[5] We will discuss more on the relation between P- and F-programs in Section 3.2.1 (p. 69).

a grammar G P_n-programs may be defined (this choice also affects which F-programs may be defined). It should also be possible to leave this unrestricted, and hence allow P-programs to be defined for every nonterminal of a grammar. Dictating and clarifying such restrictions helps to control the flexibility of a specific fragment system.

(b) *F-programs.* It must be possible to define *explicit variation points* in fragments—slots. Note that sentential forms in formal language theory terms are abstract entities that never are specified by programmers in practice. But, being able to define F-programs essentially means the possibility of specifying and practically working with *concrete* sentential forms. This can be realized by introducing new language constructs for representing unresolved nonterminals in F-programs.

2. *F-program access.* It must be possible to address slots in fragments. This will be realized via *name references*. Addressing slots via names—in conjunction with having new language constructs for representing them (see point 1b)—enables the implementation of their access in a language-agnostic fashion.

We shall address the above requirements by first describing how explicit variation points can be defined and accessed in F-programs, followed by how the definition of P-programs can be restricted.

Defining and referencing slots

Consider the sentential form of the *RL* grammar in (2.9), where ⟨*var*⟩ is a nonterminal not yet derived to terminal symbols. The nonterminal can be derived to any string representing a variable.

```
animal(X) :- tiger(⟨var⟩).                    (2.9)
```

In a grammar-based modularization approach the nonterminal in the above represents a variable point. Slots are concrete constructs that help to simulate sentential forms, more precisely the nonterminals in sentential forms. Thus, we want to introduce the possibility to program with slots in fragments. To achieve this we extend the addressed language with a new construct—the slot—for the purpose of specifying the variable points. Introducing a new construct for the purpose of modularization avoids the need to overload any existing construct for new and different purposes. Also, it is easier to separate the names of slots from the names used in programs of the underlying language (seen as a clear benefit in, for example, BETA [63, Chapter 17]). Suppose we want to make the nonterminal ⟨*var*⟩ "slotable" in the *RL* rule language, such that we can either directly specify variables in fragments, or leave variables unspecified as named variation points (slots). For this we would introduce two nonterminals (⟨*slot'*⟩ and ⟨*ident'*⟩), assumed not to previously exist in the base grammar (hence the prime in their names), along with their definitions, which can be found in Table 2.2.[6]

We will call the grammar in Table 2.2 the SLOT-*grammar*.[7] Slots are given names (via ⟨*ident'*⟩), enclosed by the tokens "«" and "»", such that they can be referred to. This particular concrete syntax is inspired by the Mjølner fragment system, but could

[6] In our realization we do not check the disjointness conditions on the grammars, but assume it to be true.

[7] For the sake of completeness we could specify ⟨*slot'*⟩ to be the start symbol of the SLOT-*grammar*.

⟨*slot'*⟩	::=	« ⟨*ident'*⟩ »
⟨*ident'*⟩	::=	STRING

TABLE 2.2: *The* SLOT*-grammar.*

be changed if something else is more suitable for a particular language (e.g. if that syntactic construct is already used for a different purpose).

To enable the use of slots for a particular base language, we transform the production rules of the base language's grammar appropriately. We can allow (representatives of) non-derived nonterminals $\langle n \rangle$ to appear as slots in fragments by a set of grammar transformations via function: $\psi : (CFG, n) \rightarrow CFG$, where n is a nonterminal of the input CFG. For a given input base grammar G, and nonterminal n, ψ is defined by the following transformation steps, resulting in grammar G':

1. Union the SLOT-grammar with G.[8] This means: Union the two disjoint sets of nonterminals, (disjoint) terminal token symbols, (disjoint) production rules, but retain G's start symbol.

2. For each production rule in G defining nonterminal n ($\langle n \rangle$ on the left-hand side), rename n to (previously non-existing nonterminal) n'. We denote the original n nonterminal n_0 and strings generated by the original nonterminal n for n_0-strings (or $L(n_0)$).

3. Introduce the new unit production rule: $\langle n \rangle ::= \langle n' \rangle$.

4. Introduce the new unit production rule: $\langle n \rangle ::= \langle slot' \rangle$.

Since the SLOT-grammar and G are disjoint wrt. their nonterminals, the only effect made by steps 1–3 is that the derivation of strings derivable by G from any nonterminal (of G) defined via n are one step longer when derived by G', via the additional unit production rule $\langle n \rangle ::= \langle n' \rangle$.

Theorem 2.1. *(*Safe slot extension*) Given CFG G and nonterminal n defined in G, let* $\psi(G,n) = G'$*. Then every string generated by G is also generated by* G'*, that is,* $L(G) \subseteq L(G')$.

Proof. Let G and $G' = \psi(G,n)$ be CFGs. All strings in $L(G)$ are derived the same way wrt. G', save those derived via n, so we only have to concern ourselves with such strings. Let $l \in L(G)$ be a string derived via n (indicated by having n within brackets over the derivation arrow):

$$S \xrightarrow[G]{*} \gamma_1 \xrightarrow[G]{1(n)} \gamma_2 \xrightarrow[G]{*} l$$

where S is the start symbol of G and γ_1, γ_2 some sentential forms. Now, assume the opposite of what we are trying to prove, that $l \notin L(G')$. Looking at the definition of ψ which defines G' we notice that we can just as well derive l with G' using only a single extra step via n':

[8] If ψ is applied to the same base grammar more than once, this step is only performed the first time.

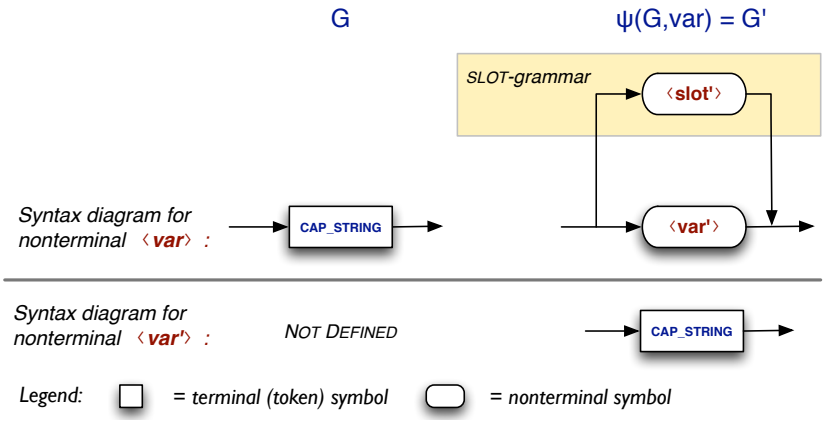


FIGURE 2.2: *To make nonterminal* ⟨var⟩ *"slotable", it is defined to be either a* ⟨var'⟩ *(the original* ⟨var⟩*), or a slot (*⟨slot'⟩*).*

$$S \xrightarrow[G']{*} \gamma_1 \xrightarrow[G']{1(n)} \gamma_2 \xrightarrow[G']{1(n')} \gamma_3 \xrightarrow[G']{*} l$$

The assumption we made was false, and hence $l \in L(G')$. □

Transformation step 4 makes the n nonterminal not only derive n_0-strings, but also n-slots: representatives for non-derived n_0 nonterminals.[9] Thus, if we let grammar G be the *RL* grammar from Example 2.1, and we apply $\psi(G, var) = G'$, then the transformed grammar G' also generates the string (2.10). The transformation is illustrated using a syntax diagram notation in Figure 2.2.

```
animal(X) :- tiger( « myVarSlot » ).
```  (2.10)

In this case, the above represents the sentential form where the slot « `myVarSlot` » represents the nonterminal ⟨*var*⟩ from G that has not yet been derived. This is, however, a piece of code that can be parsed and worked with in practice. From the perspective of grammar-based modularization, the slot represents an explicit variation point. The variation point can be referred to via its name: `myVarSlot`.

While a slot must be given a name for identification, in contrast to the Mjølner fragment system we do not require the specification of a syntactic category with the slot declaration. We shall come back to this issue in Section 2.3.

The grammar transformation via ψ enables programming with slots by explicitly introducing new constructs into the base grammar. The benefit of introducing new constructs for this purpose is that this can be done without much consideration of the particular base grammar. That is, the step can be automated. Furthermore, from a tooling perspective, the identification of slots is made simpler by having explicit constructs to represent them.

[9] Multiple applications of ψ for different nonterminals can potentially cause parsing problems due to nondeterminism, but we do not further discuss this here. Rather we assume that such problems can be resolved, either automatically, or manually.

Controlling definable fragments

To control end-users of a grammar-based modularization system, it should be possible to restrict which kind of fragments may actually be defined. That is, for which $n \in N$ of grammar G F_n^G-programs may be defined. (We here see F-programs as generalizations of P-programs, so the same restrictions would hold for P-programs.) Consider yet again our example rule language grammar from Example 2.1. Say we want to be able to define programs, atoms and variables as fragments, but nothing else. This means that we want to be able to handle strings generated by the nonterminals $\langle prgm \rangle$, $\langle atom \rangle$ and $\langle var \rangle$, respectively. This can be accomplished by changing the start symbol of the grammar depending on which fragment type should be considered. For example, by specifying the start symbol to be $\langle atom \rangle$, we have a grammar that can only generate *atom*-strings. For a given grammar G, declaring what kind of F-programs may be defined can be achieved by specifying a subset $F \subseteq N$ of the nonterminals N of G.

We want to formalize how a context-free grammar describing some language can be adapted, or transformed, to be useable in a grammar-based modularization approach. That is, formalize the steps to enable the possibility of (i) defining fragments, (ii) defining explicit slots in fragments and (iii) restricting what kind of fragments may be defined. These issues have been addressed and discussed separately in the above, but here we formally define how these steps can be brought together.

We recall that the SLOT-grammar defines the nonterminals $\langle slot' \rangle$ and $\langle ident' \rangle$ (assumed to be disjoint from any nonterminal set in any addressed grammar), and is reusable for any grammar adaptation. The SLOT-grammar, together with the application of the function ψ, and the specification of a subset of nonterminals from the base grammar to restrict the specifiable fragment types, essentially make up the adaptation. We call such an adapted grammar for a *context-free reuse grammar*, however, initially without considering the restriction of different fragment types.

Definition 2.3. *(*Context-free reuse grammar*) Let $I_{slot} = (N_s, \Sigma_s, P_s, S_s)$ be the* SLOT*-grammar, and $G = (N, \Sigma, P, S)$ a base CFG to be adapted. Given a set $N_{slot} \subseteq N$ of nonterminals representing constructs for which we want to be able to define slots, we apply the following steps:*

1. *Construct the grammar $G' = (N \cup N_s, \Sigma \cup \Sigma_s, P \cup P_s, S)$, from the* SLOT*-grammar I_{slot} and the base grammar G.*
2. *For pairwise different $i_1, \ldots, i_n \in N_{slot}$, where $|N_{slot}| = n$, apply:*

$$\psi(\psi(\ldots\psi(G', i_1)\ldots, i_{n-1}), i_n) = G''$$

Then, the grammar G'' is a context-free reuse grammar.

Notice that the start symbol of the SLOT-grammar I_{slot} is of no interest. Also notice that the reuse grammar has the start symbol S of G as its start symbol. But, because of our need to be able to handle several different fragment types, a grammar adaptation actually results in a *family* of grammars, where each family member only differs in the start symbol used. The size of the family generated from one particular base grammar is equal to the number of desired valid fragment types.

Definition 2.4. *(*Context-free reuse grammar family*) Let $R = (N_r, \Sigma_r, P_r, S)$ be a context-free reuse grammar as defined in Definition 2.3, and $G = (N, \Sigma, P, S)$ the base CFG from which R was derived. Given a set $N_{frgmt} \subseteq N$ of desired fragment types, where*

$s_1, \ldots, s_n \in N_{frgmt}$ *are pairwise different and* $|N_{frgmt}| = n$*, we define a* context-free reuse grammar family *as:*

$$\mathcal{G} = \{(N_r, \Sigma_r, P_r, s_1), \ldots, (N_r, \Sigma_r, P_r, s_n)\}$$

We can also write such a grammar family as: $\mathcal{G} = (N_r, \Sigma_r, P_r, \{s_1, \ldots, s_n\})$. The appropriate grammar in a reuse grammar family is used to handle the appropriate fragment type. As such, a context-free reuse grammar family captures our full adaptation to a grammar-based modularization approach for a given base context-free grammar.

Definition 2.4 is not a new or strange thing to a formal language theorist, since CFGs are sometimes defined in this way directly. And in practice we never have to generate a family of grammars since modern parsing libraries and systems (e.g. ANTLR [72]) allow to dynamically specify the start symbol when parsing a piece of source code. A context-free grammar G generates a (possibly infinite) set of strings $L(G)$. An analogue set can be defined for a context-free reuse grammar family: a context-free grammar family $\mathcal{G} = \{G_1, \ldots, G_n\}$ generates a (possibly infinite) set of strings defined by $L(\mathcal{G}) = L(G_1) \cup \ldots \cup L(G_n)$.[10] A string in $L(\mathcal{G})$ is called a *well-formed sentence*, or *valid sentence*, wrt. $\mathcal{G}$. When there is no confusion we will refer to a reuse grammar family $\mathcal{G}$ simply as a reuse grammar G (that may have several start symbols).

The reason we highlight the grammar family is that it formally captures—from a grammar-based modularization and composition point of view—what a grammar adaptation to GBM constitutes in our approach. We observe:

> If the reuse grammar $\mathcal{G}$ is derived from base grammar G, then $\mathcal{G}$ dictates the valid GBM components for $L(G)$ – both their valid structure and interfaces.

Thus, $\mathcal{G}$ dictates both how valid modules look (strings of a certain form), but also how they may be accessed (specified using the slot construct).

Example 2.2. (*Business rules for small company*) This example considers business rules for a small company, written in our rule language *RL* (grammar from Example 2.1). The company wants to be able to define reusable units that can be assembled in a fragment system. Suppose the following fragment is wanted to be defined, stating that employees, subject to an unspecified condition(s), receive a yet unspecified bonus:

```
bonus(X, « value ») :- employee(X), « condition ».        (2.11)
```

The above fragment can be reused by binding the two slots. First, by providing a qualifying condition for receiving a bonus at all (slot `condition`), and then by binding the unspecified bonus value (slot `value`). For example, by binding the slot `value` with P_{num}-program `"200"`, and by binding the slot `condition` with P_{atom}-program `"efficient(X)"`. This would result in the rule:

```
bonus(X, 200) :- employee(X), efficient(X).        (2.12)
```

The rule below could also be constructed from the one in (2.11), by binding the fragment `"100"` to slot `value` and the fragment `"overtime(X,Y), lg(Y,50)"` to slot

[10] Remember that $G_1, \ldots, G_n$ only differ in their start symbol.

`condition`:

```
bonus(X, 100) :- employee(X), overtime(X,Y), lg(Y,50).        (2.13)
```

The binary predicate `lg` in (2.13) represents the "larger than" relationship. Notice that we are able to replace the single slot `condition` with two *atom*s since they appear as part of a list construct, namely, the body list of the rule [63, pp. 263–264].

Notice that the composed rules above are valid rules of the original grammar from Example 2.1. To adapt the grammar from Example 2.1 such that the above fragments can be defined, and appropriately transformed via slots, we create a context-free reuse grammar according to Definitions 2.3–2.4. As an example, we define: $N_{slot} = \{num, atom\}$ and $N_{frgmt} = \{prgm, num, atom\}$.[11] To generate the needed reuse grammar, we apply Ψ two times (for each member of N_{slot}). The result is the grammar $G' = (N, \Sigma, P, S)$:

$\langle prgm \rangle$ ::= $\langle stmt \rangle$*
$\langle stmt \rangle$ ::= $\langle rule \rangle$ | $\langle fact \rangle$
$\langle rule \rangle$ ::= $\langle head \rangle$:- $\langle body \rangle$.
$\langle head \rangle$ ::= $\langle atom \rangle$
$\langle fact \rangle$ ::= $\langle atom \rangle$.
$\langle body \rangle$ ::= $\langle atom \rangle$ (, $\langle atom \rangle$)*
$\langle atom' \rangle$ ::= $\langle predname \rangle$ ($\langle term \rangle$ (, $\langle term \rangle$)*)
$\langle term \rangle$ ::= $\langle const \rangle$ | $\langle var \rangle$ | $\langle num \rangle$
$\langle atom \rangle$ ::= $\langle atom' \rangle$

$\langle atom \rangle$ ::= $\langle slot' \rangle$
$\langle num \rangle$ ::= $\langle num' \rangle$
$\langle num \rangle$ :: $\langle slot' \rangle$
$\langle slot' \rangle$ ::= « $\langle ident' \rangle$ »
$\langle ident' \rangle$ ::= STRING
$\langle predname \rangle$ ::= STRING
$\langle const \rangle$ ::= STRING
$\langle var \rangle$ ::= CAP_STRING
$\langle num' \rangle$ ::= NUM_STRING

The grammar $G' = (N, \Sigma, P, \{prgm, num, atom\})$ is created from N_{frgmt} to complete the adaptation. To better demonstrate how this grammar encodes our intensions, we split it into a grammar family:

$$\mathcal{G}' = \{G'_1 = (N, \Sigma, P, prgm), G'_2 = (N, \Sigma, P, num), G'_3 = (N, \Sigma, P, atom)\}$$

We give three examples of how the above fragments do (not) belong to the language generated by $\mathcal{G}'$:

1. F_{prgm}-program "`bonus(X, «value») :- employee(X), «condition».`" can be generated from G'_1 (occasionally skipping derivation steps indicated by the number above the derivation arrow):

   <prgm> $\xrightarrow{2}$ <rule> $\xrightarrow{1}$ <head> :- <body>. $\xrightarrow{4}$ bonus(<var>, <term>) :- <body>. $\xrightarrow{2}$ bonus(X, <num>) :- <body>. $\xrightarrow{1}$ bonus(X, <slot'>) :- <body>. $\xrightarrow{5}$ bonus(X, «value») :- <atom'>, <slot'>. $\xrightarrow{*}$ bonus(X, «value») :- employee(X), «condition».

2. From G'_3 the P_{atom}-program "`efficient(X)`" can be generated:

[11] The potential relation between the sets N_{slot} and N_{frgmt} is discussed in Section 3.2.1 (p. 69).

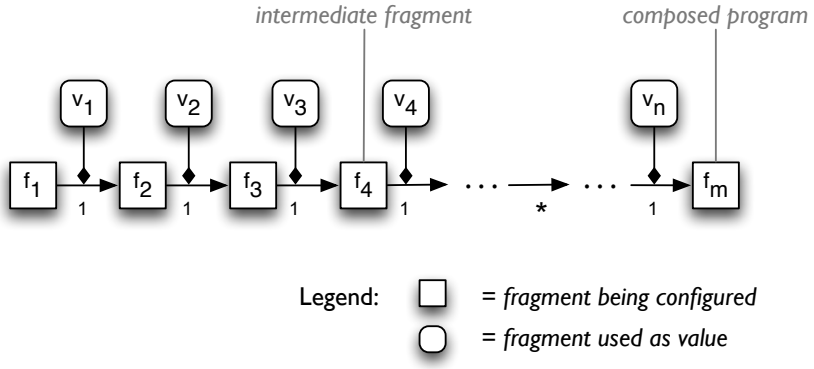


FIGURE 2.3: *The composition process can be seen as a* composition chain *where each link in the chain—binding fragments to slots—contributes to the final result.*

<atom> $\xrightarrow{1}$ <atom'> $\xrightarrow{1}$ <predname>(<term>) $\xrightarrow{2}$ efficient(<var>) $\xrightarrow{1}$ efficient(X)

3. The P_{var}-program `"Z"` cannot be specified and should produce an error in the fragment system. This is because none of the grammars in $\mathcal{G}'$ can generate the string `"Z"`.

■

In the above we have described how a language, in particular its grammar, can be adapted such that it may be used for grammar-based modularization. This includes defining what appropriate fragments are and exactly where slots may appear in such fragments.

2.2.2 Generic fragment language – FL^{ABS}

In a grammar-based modularization system, the combination of fragments into the final result often takes a number of *composition steps*, resulting in a composition chain (illustrated in Figure 2.3). Each such step is a *slot application*, that is, the binding of a slot with a fragment. The exact fragments being put together—which slot applications to carry out—must be specified by a developer using some formalism, or *fragment language*. Programs of the fragment language are called *composition programs*. By analyzing the specialized fragment language from the Mjølner fragment system, the following fundamental fragment language requirements can be identified:

1. *Fragment declaration.* It must be possible to define and declare fragments that should be used in the composition process.

2. *Slot application.* The language must support the basic fragment assembly technique: binding fragments to slots. That is, the notion of slot application must be available as a construct.

3. *Result specification.* It should be possible to specify which fragment is considered the result of the composition.

The above represents essential constructs for any fragment language for grammar-based modularization. The Mjølner fragment system itself has its own special way of supporting the above requirements. Fragment-forms define and declare fragments. Slot applications are specified using the *origin* construct and by using the same name for a fragment-form as the slot to which it is intended to be bound. Slot applications are thus specified implicitly using name convention (or merging of names during composition). Result specification is also done implicitly: there exists one fragment-form on which no other fragment-form depends, and this fragment-form makes up the result of the composition. This 'output' fragment-form often contains boiler-plate code required to properly setup the BETA execution environment.

In our framework approach we want to use a single fragment language irrespective of the underlying grammar. For such an approach to work, information that cannot be presupposed in the fragment language must instead be given in the composition programs. This includes which grammar (language) is being worked on and what kind of fragments are being declared. In the following our goal is to use the above identified fragment language construct requirements to define the foundations for a generic fragment language. The defined fragment language is generic in the sense that it does not care about the language used to write the actual fragments. The goal here is to define a simple language, using intuitive concrete syntax, which should be seen as a template or straw man for a more concrete realization and implementation. For this reason, we refer to the language as being 'abstract', and call it FL^{ABS}. The superscript *ABS* of the language name symbolizes this. The fragment language can concretely be realized in any existing programming language, as long as it supports the same constructs as described in the abstract language. For example, it can be implemented as an API, or designed as a standalone language.

Declaring a fragment of our example rule language *RL* (from Example 2.1) of syntactic category *var* can look like in (2.14).

```
fragments (rl,var) myfrag = file:myvar.var .        (2.14)
```

The declaration in (2.14) associates the name `myfrag` with the fragment(s) in the file `file:myvar.var`. Notice how the type of the fragment is declared: both a grammar (`rl`) and a specific syntactic category (`var`) from that grammar is given. That is, the "types" of fragments in this fragment language are tuple types: *grammar* $\times$ *syntactic category*. In practical terms this allows a system implementing the framework, where the fragment language is used, to know which grammar, and which start symbol, to use when parsing. The syntactic category used in (2.14) (*var*) must be supported by the assumed existing reuse grammar.

Abstractly a slot application can be expressed as in (2.15), where F is the *target fragment* and F' is the resulting fragment after binding *value fragment* f to the slot *slot* in F.

$$F \xhookrightarrow{(slot,f)} F' \tag{2.15}$$

Formally speaking, a slot application amounts to replacing a sequence of terminal symbols in F, that represent the slot, with the string f. The resulting string F' may, or may not, belong to the language of the considered reuse grammar. Here we only concern ourselves with the actual string replacement, but in the next section we shall discuss what a *safe* slot application can be considered to be. Also, (2.15) can be seen as an expression that evaluates to the resulting string. That is, (2.15) evaluates to F', the *result fragment*.

⟨*prgm*⟩	::=	⟨*stmt*⟩*
⟨*stmt*⟩	::=	⟨*fdecl*⟩ | ⟨*bind*⟩ | ⟨*print*⟩
⟨*fdecl*⟩	::=	`fragments (` ⟨*ident*⟩ `,` ⟨*ident*⟩ `)` ⟨*ident*⟩ `=` ⟨*ref*⟩ `.`
⟨*bind*⟩	::=	`bind` ⟨*ident*⟩ `in` ⟨*ident*⟩ `with` ⟨*ident*⟩ `.`
⟨*print*⟩	::=	`print` ⟨*ident*⟩ `to` ⟨*file*⟩ `.`
⟨*ref*⟩	::=	⟨*file*⟩ | ⟨*inline*⟩
⟨*inline*⟩	::=	`"` ⟨*text*⟩ `"`
⟨*ident*⟩	::=	STRING
⟨*file*⟩	::=	LOCATION
⟨*text*⟩	::=	TEXT

TABLE 2.3: *Grammar for the generic fragment language* FL^{ABS}.

Consider the fragment F in (2.16). Let us call fragment `"200"` for f and fragment `"efficient(X)"` for g.

```
bonus(X, « value ») :- employee(X), « condition ».        (2.16)
```

Then the resulting fragment F'' of the abstract sequence of slot applications in (2.17) is as expected (cf. (2.12)).

$$F \xhookrightarrow{(value,f)} F' \xhookrightarrow{(condition,g)} F'' \tag{2.17}$$

In this case the resulting string F'' is a valid *RL* rule. However, the slot application sequence in (2.17) results in an invalid *RL* statement if we change the value fragment g to `"X"`, since this would create the string in (2.18).

```
bonus(X, 200) :- employee(X), X.        (2.18)
```

We introduce this abstract notation because it can be convenient to use when explaining slot applications, and we will make use of it later. More concretely we can write slot applications (cf. (2.15)) as shown in (2.19), where `slot` is the name of a slot in `F` and `f` is a declared fragment.

```
bind slot in F with f .        (2.19)
```

For specifying which fragment is considered the result of the composition program, the `print` construct can be used:

```
print frag to file:out.prgm .        (2.20)
```

where `frag` is the resulting fragment and `file:out.prgm` the desired output file. The grammar of our generic fragment language FL^{ABS} can be found in Table 2.3. The first production rule in the grammar in Table 2.3 says that composition programs consist of a sequence of statements. A statement is either a fragment declaration (⟨*fdecl*⟩),

```
1 fragments (rl,rule) myrule = file:rprgm.rule .
2 fragments (rl,num) bonus = "200" .
3 fragments (rl,atom) cond = "efficient(X)" .
4
5 bind value in myrule with bonus .
6 bind condition in myrule with cond .
7
8 print myrule to file:outprgm.rule .
```

LISTING 2.1: *Simple composition program composing three fragments.*

a slot application (⟨*bind*⟩), or a print statement (⟨*print*⟩). Declarations reference fragments (defined by nonterminal ⟨*ref*⟩) either via a file location (⟨*file*⟩) or an inline fragment definition (⟨*inline*⟩). An inline fragment definition means that a string containing the fragment is specified (see Lines 2–3 in Listing 2.1).

Example 2.3. (*Simple composition program*) We revisit the rule in (2.21) written in an extension of our rule language (referred to as *rl*). Let us assume it is located at `file:rprgm.rule`.

```
bonus(X, « value ») :- employee(X), « condition ».        (2.21)
```

Assume we want to compose the program in (2.22).

```
bonus(X, 200) :- employee(X), efficient(X).        (2.22)
```

The composition program in Listing 2.1 would achieve this. The first statement in Listing 2.1 declares the fragment in (2.21). The second and third statements define and declare two inline fragments. The two bind statements then realizes the simple composition.

■

The above fragment language is able to declare fragments and put them together using sets of slot applications. It is generic by not making assumptions on the language used to define fragments. This genericity is achieved at the expense of forcing programmers to precisely declare which grammar is considered, as well as which syntactic categories are used.

2.3 Grammar types and safe slot applications

When constructing software from smaller parts (components), it is important to be able to provide some kind of guarantees wrt. the correctness of the resulting software. There are two well-known levels of correctness that can be distinguished. We here briefly discuss them and their differences, particularly in the setting of grammar-based modularization. In this work we will mainly focus on the first.

1. *Context-free syntactical correctness.* The most basic kind of guarantee that we would like to provide when composing software fragments is that the resulting program is syntactically correct wrt. the underlying language. This means that

the resulting program belongs to the language $L(G)$ generated by the base grammar G. If this is the case, not only do we have a possibly meaningful program, but the composed programs can also be used and understood by existing tools developed for the underlying language, such as editors, parsers, interpreters, compilers, reasoners, analyzers.

2. *Context-sensitive syntactical correctness.* Ideally, one would also be able to guarantee other properties of the composed program. For example, that it not only is syntactically correct, but that it also will compile (if it is to be compiled) or execute (if it is executable) as expected or without errors. Such properties have to do with the semantics of the program. It could for example be checked—statically (without executing or interpreting the program)—that a variable has been defined before it is used. Informing the programmer of such errors is clearly very helpful.

In this work we consider the first case from above, context-free syntactical correctness. The second case is outside the scope of this work. Context-sensitive syntactical correctness is first of all very dependent on the particular language in which the fragments are written, which makes it hard to deal with in a generic manner. It is also difficult to make such guarantees during composition of fragments since it is not well-defined when the checks are to be performed (for certain fragment transformations they might not make sense). It can be checked when the composition is complete, but then it is difficult to know which exact composition step caused the error.

In a grammar-based modularization system, software is constructed by composing source code fragments. In general we want to guarantee that the final composed program is a well-formed program of the underlying language. In particular we want to ensure that every single composition step is a *safe* and valid one. Not giving such guarantees would enable programmers to construct useless software. In our generic approach it is important that safeness guarantees can be handled, or formalized, independently of any particular language instantiating our framework. Or rather, we need a general understanding of how a base grammar can be used to construct the required safety conditions. We will define safeness guarantees wrt. the formalism on which our framework is based, namely, context-free grammars and the languages they generate.

2.3.1 General safeness conditions

Before we give general safety conditions for compositions we look at a simple example.

Example 2.4. (*Syntactically correct compositions*) Consider *RL*'s grammar from Example 2.1 extended by allowing *atom*s to be "slotable", and *prgm*s, *atom*s and *var*s to be defined as fragments. Then the following fragment can be defined:

```
« consequence » :- tiger(X).
```
(2.23)

Binding the slot named `consequence` with P_{var}-program `"Y"` should render an error since the resulting program is not in the language generated by neither the base grammar, nor the reuse grammar. Thus (where $\hookrightarrow$ reads "composes into", and $\not\hookrightarrow$ reads "cannot be composed into"):

```
« consequence » :- tiger(X).   ↪   Y :- tiger(X).
```
(2.24)

However, binding the same slot with the P_{atom}-program `"mammal(X)"` should be allowed, since the resulting fragment would be in the language generated by the base grammar. That is:

```
« consequence » :- tiger(X).  ↪  mammal(X) :- tiger(X).    (2.25)
```

Notice that when only ensuring context-free syntactical correctness, the following would also be allowed:

```
« consequence » :- tiger(X).  ↪  mammal(Y) :- tiger(X).    (2.26)
```

The resulting program is a syntactically well-formed sentence of the rule language, but it is not *semantically* sound since the variable used in the rule head does not appear in the rule body. Such semantical considerations could be specified in the static semantics of the language, but, as mentioned, we do not cover such safety conditions. ■

Intuitively, for any composition result to be syntactically correct wrt. the base language (that is, to be a string generated by the base grammar), each fragment bound to a slot must belong to the set of strings generated by the nonterminal for which the slot is a representative. That is, there is a clear connection between slots simulating sentential forms and fragments replacing such slots. In a grammar-based modularization approach, context-free syntactical correctness of composition results can be guaranteed by referring to the underlying grammar.

To clarify and formalize our intuition we introduce the notion of *grammatical types*, both for fragments and slots, which make up the formalism for our notion of *safe* compositions. Whenever we refer to the *type* of a fragment or a slot, we shall mean its grammatical type.

Definition 2.5. *(*Grammatical types for fragments*) Given a context-free grammar* $G = (N, \Sigma, P, S)$*, a context-free reuse grammar* G' *derived from G, and a fragment F, every nonterminal* $n \in N$ *that F can be derived from (wrt.* G'*) is a* grammatical type *of F, and its set is denoted* $\tau(F)$*.*

$$\tau(F) = \{n \in N \mid n \xrightarrow[G']{*} F\}$$

As can be noticed by Definition 2.5, a fragment can have several types. The reason we use the reuse grammar G', rather than base grammar G, for deriving F is that we also have to consider fragments with slots (not possible if we derive wrt. G). While G' is used for deriving F in the definition, notice that only members of N (of G) are considered types, not the nonterminals of G'. The SLOT-grammar nonterminal $\langle slot' \rangle$, for example, is never considered a grammatical type. When no reuse grammar is under consideration, then the same definition holds using the base grammar G as a condition for deriving fragments (see Example 2.5).

Example 2.5. Consider the rule language *RL*'s grammar from Example 2.1, call it G, and the following fragment F:

```
tiger(sherekhan).    (2.27)
```

Then F has three grammatical types wrt. G, namely, $\tau(F) = \{prgm, stmt, fact\}$. All these nonterminals of G can generate F (F is a program, a statement, and a fact).

■

Often a fragment will be declared a specific, single, type by a programmer. If the type set $\tau(F)$ for a fragment F is required, it can be computed using, for example, the CYK[12] algorithm [56]. Other more efficient algorithms could also be used, but we do not further investigate this here. Next we define grammatical types for slots.

Definition 2.6. *(*Grammatical types for slots*) Assume a context-free reuse grammar $\mathcal{G}' = (N', \Sigma', P', \{s_1, \ldots, s_p\})$ derived from the base grammar $G = (N, \Sigma, P, S)$, a fragment F well-formed wrt. $\mathcal{G}'$, and a slot V in F. A nonterminal $n \in N$, generating a set of strings $L(n)$, where each string in $L(n)$ can replace V in F while keeping the result F' in $L(\mathcal{G}')$ is a* grammatical type *of V. That is, $n \in N$ is a* grammatical type *of V, if:*

$$\forall f \in L(n) : (F \xhookrightarrow{(V,f)} F') \in L(\mathcal{G}')$$

The set of all such nonterminals is called the grammatical types of V, and its set is denoted $\tau(V)$:

$$\tau(V) = \{n \in N \mid \forall f \in L(n) : (F \xhookrightarrow{(V,f)} F') \in L(\mathcal{G}')\}$$

An example illustrates the definition.

Example 2.6. Consider the rule language *RL*'s grammar from Example 2.1, call it G, and its reuse extension G'. Assume the following fragment F to be valid wrt. G':

```
bonus(X, « value ») :- employee(X).
```  (2.28)

Then the slot `value` has four grammatical types: $\tau(value) = \{term, var, const, num\}$. Any string generated by these nonterminals can replace `value` while keeping F valid wrt. G' (and in this case also G). This essentially means that the slot is dynamically typed by the base grammar; the appropriate type can be assumed for the slot depending on the exact type of an actual fragment being bound to it.

A word of warning should be mentioned here. The fragment in (2.28) is assumed to belong to the language generated by G'. However, for this particular fragment, the needed extension G' can be derived from G by applying ψ in several different ways. Suppose the person creating the reuse grammar wanted to make variables "slotable" and performed $\psi(G, var) = G'$. Fragment (2.28) can now be authored, but according to Definition 2.6 the slot is associated with several types and not only the expected type *var*. We shall come back to a discussion on this seeming conundrum later in this section.

■

As mentioned, the composition of some software artifact can take a number of steps, where each step transforms the involved software entities is some particular way (see the composition chain in Figure 2.3). In our grammar-based modularization framework each such step is a slot application, for which we would like to have some guarantees of its correctness. Given a base grammar G and its reuse grammar G', our safety conditions should guarantee that:

[12] Cocke-Younger-Kasami

S1 Each building block (each fragment used) should be a well-formed sentence wrt. G'.

S2 For each step i in a composition chain (see Figure 2.3), given that the fragment f_i is a well-formed sentence wrt. G', fragment f_{i+1} is also a well-formed fragment wrt. G'.

S3 The last fragment in a composition process (chain), the composition result, must not only be a well-formed sentence wrt. reuse grammar G', but also a well-formed sentence wrt. its corresponding base grammar G.

The first safety condition holds for a fragment F if $F \in L(G')$ where G' is the context-free reuse grammar under consideration. We shall assume that this always holds for fragments. Below we will deal with ensuring that every fragment resulting from a slot application results in a valid sentence of the reuse language. Later we shall come back to the last safety condition.

Definition 2.7. *(*General type safety*) Let G' be a context-free reuse grammar, F and f fragments valid wrt. G', and V a slot in F, then:*

$$F \xhookrightarrow{(V,f)} F' \quad \textit{if and only if} \quad \tau(V) \cap \tau(f) \neq \emptyset$$

That is, the composition step (slot application) is safe if and only if the slot and the fragment replacing the slot have at least one common type. Notice that Definition 2.7 is a very general notion of a safe slot binding, which hardly would be used in practice. This since it is not easy (if possible at all) to calculate the set $\tau(V)$. And even though the set $\tau(f)$ can be calculated, it might not always be desirable. However, the condition intuitively formalizes a safe composition step, and we will make use of this definition in more restricted cases, where we do not have to calculate the type sets. In practical settings, we will do one of the following:

1. Execute $F \xhookrightarrow{(V,f)} F'$ and if $F' \notin L(G')$ where $L(G')$ is the reuse language, the composition step results in an error.
2. The type(s) of the slot V and the value fragment f slot can be declared, and in this case we can check the condition in Definition 2.7 (we shall see cases like this below).

Example 2.7. (*Slot application safety*) Consider our rule language *RL*'s grammar from Example 2.1, call it G, and its reuse extension G'. Assume the fragment (2.29), call it F, is valid wrt. G'.

```
bonus(X, « value ») :- employee(X).        (2.29)
```

The types for the slot `value` is $\tau(value) = \{term, var, const, num\}$. Suppose we want to bind the slot `value` with the G'-valid P-program `"200"`, call it f:

$$F \xhookrightarrow{(value,f)} F' \tag{2.30}$$

The types of f are $\tau(f) = \{term, num\}$. Since $\tau(f) \cap \tau(value) = \{term, num\} \neq \emptyset$ this composition step is safe. The result F' is shown in (2.31).

```
bonus(X, 200) :- employee(X).        (2.31)
```

Binding the same slot with P-program `"efficient(X)"`, call it g, would result in an error. This is because $\tau(g) = \{atom, head\} \cap \tau(value) = \emptyset$.

■

The above has explained and demonstrated general composition safety, where the safety conditions are based on grammar types from the considered language grammar. The conditions are defined in terms of the standard semantics of context-free grammars, that is, the languages they generate. Every safe slot application keeps the result in the language generated by the considered reuse grammar. In this general setting slots are not explicitly associated with syntactic categories, rather implicitly via the context-free grammar semantics. This is a difference from the approach taken by the Mjølner system where slots are always defined with a syntactic category (type).

In Definition 2.6 we are strict in our definition of which nonterminals are considered types for slots. Only a nonterminal n whose every language member (that is, is in $L(n)$) may syntactically replace the slot is considered a type. So, if only a subset $L(n)^- \subseteq L(n)$ syntactically may replace the slot, n is not considered a type. The safety condition in Definition 2.7 is in this sense an over-approximation. This simply means that if the safety condition was to be checked statically, certain non-erroneous slot applications would be disallowed. We explain by an example.

Example 2.8. Consider the *RL* fragment below:

```
«fact» .                                                    (2.32)
```

The types of the slot `fact` are $\{head, atom\}$. However, the nonterminal $\langle body \rangle$ may also generate strings that could be bound to `fact` while retaining the result valid wrt. *RL*'s grammar. But *body* is not considered a type of `fact` since $\langle body \rangle$ also generates strings that cannot replace `fact` while keeping the result valid wrt. *RL*'s grammar. The string `"atom1(a), atom2(b)"` is an example.

■

At times it can be desirable to give programmers more control over what is considered "safe" for their fragments. Consider, for example, the fragment from (2.28). From our intuitive understanding of the example it is clear that the author of that fragment does not intend for a variable to be bound to the slot `value`. Rather, the author clearly intends for a number (the bonus amount) to be bound to the slot. Nonetheless, the general typing conditions stated above would permit a variable to be bound to the slot. The reason for this is that the exact transformation of the base grammar via ψ is not considered by the safety condition. (For example, in this case, there is no difference between transforming the base grammar G by $\psi(G, var)$ or $\psi(G, term)$.) Instead, the safety condition so-far specified only ensures syntactical correctness throughout the composition process. We would hence like to enable fragment authors to be more precise in how they expect their fragments to be transformed during composition. This can essentially be done in two ways:

1. *User-restricted slot applications.* Allow fragment authors to associate a slot with an expected fragment type (syntactic category).
2. *Syntax-restricted slot applications.* Provide specific syntax for each nonterminal representative (slot type).

In the first case the type of the slot is declared by the user, while in the second case the slot type is mandatory and implicitly chosen depending on the particular slot construct that is being used (in this case several different slot constructs might be available). We will consider both options below, followed by summary of their advantages and disadvantages.

⟨*slot'*⟩ ::= « ⟨*ident'*⟩ (: ⟨*ident'*⟩ (, ⟨*ident'*⟩)*)? »

TABLE 2.4: *Alternative slot construct with possibility to constrain slot applications.*

2.3.2 User-restricted slot applications

Giving fragment programmers the possibility to explicitly specify a desired type for a slot would give authors more control over how their fragments are used. Imagine that the fragment author would specialize the fragment from (2.28) into the one in (2.33).

```
bonus(X, « value :  num ») :- employee(X).
```
(2.33)

The added information to the declared slot in (2.33) would restrict fragments transforming the slot `value` to belong to the syntactic category *num*. The safety condition as defined in Definition 2.7 still holds. Notice that in general a programmer could associate not only one, but several types with a slot.

Example 2.9. Consider the rule language *RL*'s grammar from Example 2.1, call it G, and its reuse extension G'. Assume that the fragment (2.33), call it F, is valid wrt. G'. The types for the slot `value` is, according to the specified type restriction, $\tau(value) = \{num\}$. Suppose we want to bind the slot `value` with the G'-valid *P*-program `"200"`, call it f. The types of f are $\tau(f) = \{term, num\}$. Since $\tau(value) \cap \tau(f) = \{num\} \neq \emptyset$, the composition step in (2.34) is safe.

$$F \xhookrightarrow{(value,f)} F' \tag{2.34}$$

Trying to bind `value` with *P*-program `"X"` on the other would be disallowed since $\tau(value) \cap \{term, var\} = \emptyset$.

■

One condition on such slot annotations is that the specified type(s) must be in the set of the slot's general types. Consider the fragment in (2.33). The explicitly specified type for the slot `value` must belong to the set $\{term, var, const, num\}$, the general types for `value` (see Definition 2.6).

Definition 2.8. *(*Type annotated slots*) A* type annotated slot *is a slot with an explicitly specified type. For a slot V, its annotated type is denoted $\downarrow \tau(V)$ and is subject to the condition:*[13]

$$\downarrow \tau(V) \subseteq \tau(V)$$

Concretely allowing fragment programmers to restrict the types of slots can be achieved by redefining the ⟨*slot'*⟩ nonterminal in the SLOT-grammar. This is done in Table 2.4, where the declaration of a specific type(s) is optional.

The kind of restrictions discussed above can be extremely useful for programmers. However it is still possible to transform the *RL* grammar G by $\psi(G, var) = G'$ and restrict a slot by *num* as in (2.33). But, as long as Definition 2.8 is honored the restriction desired by the programmer is achieved. Still, as can be seen, there is somewhat

[13] It makes sense to sharpen this restriction to: $\downarrow \tau(V) \subseteq \tau(V) \cap N_{slot}$ where N_{slot} is the set of nonterminals used to create the considered reuse grammar. This since the restriction should also respect the reuse grammar.

⟨*n-slot'*⟩ ::= « ⟨*ident'*⟩ : `#n#` »

TABLE 2.5: *Special slot construct for a nonterminal* ⟨n⟩ *of some base grammar.*

of an anomaly wrt. the grammar transformation function ψ. This in particular since it seemed that the very purpose of ψ was to allow the definition of slots as representatives for very particular nonterminals of the base grammar. Instead, ψ has mainly been used to enable the definition of fragments containing slots, and not to ensure that only slots of particular syntactic categories may be defined. This impression of a gap appears because the safety conditions are defined in terms of the languages the grammars generate, and not how the initial base grammars are transformed. The benefit of such a lax coupling between ψ and the safety conditions is that the same construct—the slot—can syntactically be used for every slot without requiring one new language construct for every nonterminal representative (slot type) desired.

However, it would also be possible to maintain the grammar transformation information in fragments by hard-coding the type in the syntax. Thus, for every $n \in N$ for a grammar G on which we apply $\psi(G,n)$, we can introduce a new construct to act as n-representatives in fragments. The type of the n-specific slot is then not (dynamically) given in fragments, but always assumed to be n.

2.3.3 Syntax-restricted slot applications

Assume that a developer wants to make numbers in the *RL* rule language (denote its grammar G) variable via the explicit slot mechanism. The developer would then perform $\psi(G,num) = G'$. The definition of ψ could be changed to instead of introducing the general slot construct discussed above, introduce a *num*-specific slot construct with a pre-defined type. We refer to this modified grammar transformation as ψ^*. As an example, consider the fragment in (2.35) where the slot `value` is assumed to be defined using the newly introduced *num*-slot construct.

```
bonus(X, « value :  #num# ») :- employee(X).        (2.35)
```

Here `#num#` is assumed to be concrete syntax, part of the slot construct and not a user-defined type. Different syntax could be used if found to be more appropriate. The type $\tau(value)$ of the *num*-slot `value` is thus the set $\{num\}$, consistent with the modified ψ^*-transformation. Based on this example transformation it would not be possible to author a fragment with a slot that was intended to bind variables. A system enforcing the safety conditions would have to understand the relationship between syntax and types, but could then use the same typing condition as in Definition 2.7 to enforce them.

Generating such n-slots could be achieved by introducing n-slot constructs for each $n \in N$ for grammar G on which $\psi^*(G,n)$ is performed. Hence, the slot construct in Table 2.5 is used for each n instead of the more general *slot* construct as defined in the SLOT-grammar (where n corresponds to the particular nonterminal being considered). Other than this the transformation as defined by ψ is the same. The slots introduced by ψ^* corresponds to the slots in the Mjølner system.

Example 2.10. Consider the *RL* grammar from Example 2.1, call it G, and its reuse extension $\psi^*(G,num) = G'$. Then the following grammar rule will be in G':

$$\langle \textit{num-slot'} \rangle ::= \text{«} \langle \textit{ident'} \rangle \texttt{ : \#num\# } \text{»}$$

That is, the grammar rule from Table 2.5 has been instantiated for nonterminal $\langle num \rangle$. Then the following fragment F is valid wrt. G'.

```
bonus(X, « value :  #num# ») :- employee(X).        (2.36)
```

The types for the slot `value` is now $\tau(value) = \{num\}$. The single type *num* is associated with the slot because the fragment programmer has used the *num*-slot construct. This type exactly correlates with the grammar transformation ψ^*. The only fragments that can now be bound to the slot `value` are fragments having *num* as one of their types. Notice that having only performed the single transformation $\psi^*(G, num) = G'$, we can only formulate slots of the kind in (2.36) (only the slot name can be changed).

■

Summary

The above has considered and discussed different levels of granularity wrt. how grammars are transformed into reuse grammars, and how the safety conditions deployed during composition of fragments relate to such reuse grammars. Using a single slot construct, even for different nonterminal representatives, gives flexibility to programmers as they do not need to specify syntactic categories with each slot declaration. The possibility of restricting slot applications using slot annotations is still left open as an option. However, if the exact grammar transformations via ψ are desirable to be reflected in fragments, then the more specific transformation function ψ^* can be deployed. The main difference is whether the safety conditions should be dictated by developers creating the reuse grammars—*statically*—or whether these conditions are more *dynamic*, in the sense that they are loosened from the reuse grammars and certain responsibility is left to fragment users and programmers. Regardless of the preferred approach, it should be stressed that the safety conditions are always formally defined wrt. the languages generated by the considered grammars: Any intermediate composition result must belong to the set of strings generated by the reuse grammar.

BETA essentially uses syntax-restricted slot applications in its approach, but for a fixed and predefined set of nonterminals from the BETA grammar. Moreover, BETA's type restrictions are based on name matching between the specified syntactic categories of the fragment-forms and their slot applications. We instead define the restrictions wrt. the languages generated by the considered grammars.

Safety conditions on composition results

It is not only important to be able to guarantee certain form of the composition result at each subsequent step through the composition process, it is also important to have some guarantee on the form of the final composition result. We add the following to the safety requirement list.

S3 The last fragment in a composition process (chain), the composition result, must not only be a well-formed sentence wrt. reuse grammar G', but also a well-formed sentence wrt. its corresponding base grammar G.

See fragment f_m in Figure 2.3 for an illustration. Being able to guarantee that the final composition result is a sentence of the underlying base grammar allow for existing tools to work with the result (compilers, interpreters etc.). If a composed fragment F is a sentence of the base grammar, we call F for *bound*.

Definition 2.9. *(*Composition result safety*) Let G be a base grammar, G' a context-free reuse grammar derived from G and F_i a fragment well-formed wrt. G' (during some step i of the composition process). If F_i belongs to the language generated by G, then F_i is said to be* bound*. That is, if:*

$$F_i \in L(G)$$

The interface of a *bound* fragment is "exhausted." It should be noted that in general the exhaustion of a fragment can be the result of binding all its slots, or possibly by removing unbound slots (if allowed by the base grammar).

2.4 Summary

Let us recap what we accomplished in this chapter. We started from the desire to develop a universal and lightweight approach to grammar-based modularization. That is, an approach able to address arbitrary languages, albeit with certain limitations compared to earlier approaches (e.g. no support for separate compilation of fragments, as in [63]).

- *Grammar adaptations for GBM interfaces.* We specified a simple method—captured by the function ψ—for how grammars can be adapted safely such that it is possible to author the kind of interfaces used in a grammar-based modularization approach: fragments configurable via slots.

- *Lightweight fragment systems.* We formalized the encoding of fragment systems in *reuse grammar (families)*. That is, a reuse grammar dictates the kind of fragments that may be specified in the fragment system it encodes. Towards useable fragment systems we also described the minimal requirements for a generic fragment language, summarized in the exemplary language FL^{ABS}.

- *Safe composition.* An important parts covered in this chapter was how imperative safety conditions can be given based on the base language specifications (grammars). The composition conditions ensure that the result after each composition step remains a well-formed fragment of the considered adapted grammar G', and that the final composition result is a well-formed fragment wrt. the base grammar G (where G' is a reuse grammar derived from G).

- *Alternative grammar adaptations.* Concerning grammar adaptations and safeness of compositions wrt. such adapted grammars, we discussed two different approaches:

  1. General grammar transformation (using a single *slot* construct) via ψ with two alternative options wrt. composition safety:

     (a) *General safeness conditions.* Implicit typing of slots. Slot applications must adhere to the conditions posed by the associated reuse grammar.

(b) *User-restricted slot applications.* Enables the possibility for programmers to explicitly restrict slot applications using slot annotations.

2. Specific grammar transformation via ψ^* using syntactic category specific slot constructs.

3

Universal Invasive Software Composition

Invasive Software Composition (ISC) is a composition approach that composes software by transforming and adapting source code fragments, ISC's notion of components [5]. ISC is a static approach, meaning that composition never takes place at runtime. At its foundation, ISC has many commonalities with the modularization technique discussed in Chapter 2—grammar-based modularization (GBM). From a conceptual point of view, the most striking difference is that ISC generalized GBM by not only considering explicit fragment interfaces—named locations in form of *slots*—but also *implicit* interfaces. That is, in ISC a fragment may be transformed in places not intended, or explicitly declared, by its programmer. Such implicit access can be compared to aspect-oriented programming (AOP) approaches such as ASPECTJ [52]. The AOP community essentially calls this phenomenon for *obliviousness* [30]—fragment programmers are oblivious to, not aware of, possible transformations of their programs (hopefully to their benefit).

COMPOST [87], the demonstrator system of ISC, has showcased its composition techniques and abilities on Java (we will refer to this particular system as COMPOST/J).[1] Adding another language to COMPOST's repertoire is time-consuming since support for the considered language can only be achieved through a manual process. Manual and hand-coded language adaptations are attractive in the sense that they can be highly specialized for the addressed language, but at the same time they hinder a wider and faster adoption of the composition approach. Due to the strong similarities between ISC and GBM, it would be beneficial if the contributions of Chapter 2 could be further extended and applied to ISC. This means, to provide a methodology for how ISC can be made applicable to, and work with, arbitrary languages. This would mean that the effort spent implementing the techniques of ISC for a new language could greatly be reduced. We call such an approach for *Universal ISC* (U-ISC).

The main contribution of this chapter is an extension of the GBM approach presented in Chapter 2. The extension adheres to the particulars of the more general and flexible composition approach taken by ISC. We believe several important things will

[1] Work has partly also been carried out on bringing ISC to XML (here referred to as COMPOST/XML).

be achieved:

1. We can make ISC a truly grammar-based approach. This means that we can automize the process of adapting ISC to a new language (based on its grammar), rather than having to manually tailor the adaptation for each specific language (grammar, e.g. Java's).

2. By extending the 'universal' approach to GBM from Chapter 2 to also include ISC, we can better understand the detailed relationship between the two approaches to composition.

3. By more conveniently being able to use ISC with different languages we will be in a better position to understand the limits and qualities of ISC as a composition approach, and hence be able to map out important research directions for the future.

In the same way that generalizing GBM in Chapter 2 lost some of its system-specific capabilities (e.g. separate compilation of fragments in the Mjølner system), generalizing ISC will require similar trade-offs from tailored solutions provided by COMPOST. For example, COMPOST/J can sometimes take advantage of Java's type system for finding errors in how fragments are assembled. We will not be able to achieve this because we are not tailoring a solution for Java.[2] However, the advantages gained by a more general approach allure, especially with how quickly ISC can be applied to arbitrary languages.

This chapter is structured as follows. In Section 3.1 we introduce ISC in more detail and discuss the idea of composition systems. In Section 3.2 we universalize ISC and discuss its connection to universalized GBM. In Section 3.3 we discuss what is needed to concretely develop composition systems. In Section 3.4 we give an overview of REUSEWARE/AIR: the concrete tooling that is used to develop composition systems. In Section 3.5 we give examples of two composition systems, one for our rule language *RL* and one for a subset of Java. Finally, in Section 3.6, we summarize the achievements of the chapter.

3.1 Background

Below we first give an introduction to ISC and then in Section 3.1.2 we discuss basic requirements for composing software.

3.1.1 Invasive Software Composition (ISC)

Invasive Software Composition (ISC) [5] is a static composition approach where pieces of source code (fragments) are transformed into usable programs, the composition results. The entities being composed are programs, or partial programs, of a particular language. Such entities are, in ISC terminology, called *fragment boxes* (or simply *boxes*). As an example, a box containing a Java method—a Java "method box"—is shown in Listing 3.1. This particular method (`setTimeStamp()`), when invoked, assigns the class variable `time` the current time value and prints an informative message

[2] A more COMPOST-like approach could be achieved, by paying the price of a more manual adaptation.

```
public void setTimeStamp() {
       this.time = new java.util.Date().getTime();
       System.out.println("Time set at: " + this.time);
}
```

LISTING 3.1: *A Java method box defined by an assignment and a print statement.*

```
public class Contract extends BankEntity {
       // attributes ...
       // methods ...
       Contract() { }
}
```

LISTING 3.2: *A Java class box for a bank contract (attributes and methods not shown).*

to standard output. The method cannot be used by itself (e.g. compiled by a Java compiler), but composed into a larger program it can provide certain functionality and can be reused across different Java classes and programs.

Not only method boxes may be defined and used, but also other kind of boxes that can be imagined to be useful: perhaps larger entities such as elaborate Java class boxes. Listing 3.2 shows a (potentially complex) Java class box with class name `Contract` (attributes and methods are intensionally left out). But also simpler boxes can be defined. As an example of a less elaborate box, Listing 3.3 shows a Java attribute box defining an attribute named `time` of primitive type `long`.

The kind of boxes that may be defined is governed by an associated *component model*. We shall shortly return to the issue and importance of component models. In order for boxes to be reusable, there must be methods in place for adapting the environment, that is, the context where they will be reused. ISC boxes—like any software components—need *composition interfaces* that can be exploited during reuse adaptation.

While many existing composition techniques mainly rely on only one kind of composition interface, ISC amalgamates two different kinds of interfaces: *explicit* and *implicit* interfaces. The possible *implicit interfaces* for boxes directly depend on the underlying language in which the boxes are defined. For example, if we consider the Java method box from Listing 3.1, we can imagine that it would be possible to insert another statement after the print statement. It is clear that this may not be intended by the author of the method box, but it is nonetheless possible. Hence, it is an implicit variation point of the box. Or why not insert debugging code, e.g. print statements, to be executed as soon as the method is entered and exited? In order to do this, we need to know something about the underlying language, for example that there are methods and that they contain statement lists. While this clearly is the case for Java, it might not be true for some other language. Hence, implicit interfaces directly depend on the underlying language in which boxes are written and capture how they can be adapted

```
public long time;
```

LISTING 3.3: *A Java attribute box defining an attribute named `time`, of type `long`.*

```
1 public class Contract extends genericSupertypeSuperClass {
2       // attributes ...
3       // methods ...
4       Contract() { }
5 }
```

LISTING 3.4: *A Java contract class box with unspecified super-class (a hook in ISC).*

```
1  public class CompositionProgram {
2        public static void main(String argv[]) {
3              // load a classbox, methodbox and an attributebox
4              ClassBox cBox = new ClassBox("Contract");
5              MethodBox mBox = new MethodBox("setTimeStamp");
6              AttributeBox aBox = new AttributeBox("time");
7
8              // bind super−type hook
9              cBox.findGenericSuperClass("Supertype").bind("CarRentalEntity");
10             // extend class attribute list with attribute
11             cBox.findHook("Contract.members").extend(aBox);
12             // extend class method list with method
13             cBox.findHook("Contract.members").extend(mBox);
14       }
15 }
```

LISTING 3.5: *Composition specification for composing a contract class with time-stamping capabilities using ISC.*

during composition, even if this is not mentioned or intended by the box author. However, in ISC, boxes can also be adapted to new contexts using *explicit interfaces*, called (explicit) *hooks* in ISC lingo. The explicit interfaces of boxes make plain to their users which points can, or must, be modified before reuse. Explicit interfaces thus intensionally generalize boxes, such that they can be refined, or configured, for new and different purposes.

For example, we might realize that the Java class in Listing 3.2 can be used for different kinds of contracts, not only bank contracts (e.g. car rental contracts). For this reason, we would like to specify the same contract class, but without having to, a priori, commit to a specific super-type entity. However, we want to signal users (or systems supporting the users) that there needs to be a super-class specified when the class is reused. Hence, we want to make the super-class an explicit hook, which is possible in ISC. To make super-classes recognizable as hooks in ISC, their names need to conform to the naming convention (or naming schema) for super-class hooks. In COMPOST/J this convention is `generic` + `[hook name]` + `SuperClass`. The super-class specified in Listing 3.4, for example, is a super-class hook with name `Supertype`.

One of the results from work on ISC was the distillation of two simple, yet fundamental, composition operators for boxes: *bind()* and *extend()*. These two operators comply with the observation in software composition and reuse in general of two pivotal composition and reuse styles: *parameterization* and *extension*. Hence, these two operators correspond to composition phenomena observable in almost any language, and are as such very general. When executed, these composition operators work by transforming the abstract syntax trees (ASTs) of the fragment boxes they are applied to.

```
1 public class Contract extends CarRentalEntity {
2       // attributes ...
3       public long time;
4       // methods ...
5       Contract() { }
6       public void setTimeStamp() {
7             this.time = new java.util.Date().getTime();
8             System.out.println("Time set at: " + this.time);
9       }
10 }
```

LISTING 3.6: *Composed class for car rental contracts with time stamping functionality.*

As an example, we will use the above-mentioned Java fragments to compose a usable and compilable Java class. Imagine that we want to use the generic contract class (Listing 3.4) to model car rental contracts. Furthermore, we want to be able to time stamp such contracts. We could implement this functionality directly in the contract class, but separating the two also allows to reuse the time stamping functionality in other applications. To achieve this, the method box of Listing 3.4 can be modified for reuse using both its implicit and explicit interface. The Java program in Listing 3.5, the language of choice in COMPOST/J [5] for describing compositions, details the steps needed to achieve the result.[3]

First the fragment boxes are declared such that they are accessible (Lines 4–6). On Line 9 the super-class is bound, on Line 11 the class member list is extended with the attribute box (Listing 3.3), and on Line 13 the same member list is extended with the time-stamping method (Listing 3.1). The result of this composition can be found in Listing 3.6. The resulting class now sub-classes `CarRentalEntity`, contains an attribute holding the time stamp and a method to set it. The above was a simple example not intended to show how to solve a problem not solvable by other designs or methods, but it demonstrated the use of both implicit and explicit interfaces, as well as the primitive operators *bind()* and *extend()*, which are the cornerstones of ISC.

3.1.2 Understanding composition: composition systems

In order to compose software in any given composition approach, a composition system is required [5]. A *composition system* describes a particular compositional setting and is made up of three distinct parts: a composition language, a composition technique and a component model (see Figure 3.1). The *composition language* is used to specify exactly which components should be put together, and in what way. (In GBM, the composition language is called fragment language.) The composition language is thus used to write *composition programs* (e.g. programs like the one in Listing 3.5). The *composition technique* describes how components are joined, while the *component model* describes what kind of components may be defined (what they may look like) and how they are allowed to be accessed and transformed during composition (their interfaces).

In general, different composition languages may exist and be used, but their existence is crucial since it must be possible for programmers to detail how reusable units are put together. A component model is essential to a composition system since it is the

[3] Certain details of the specification have been left out for space and comprehensibility reasons.

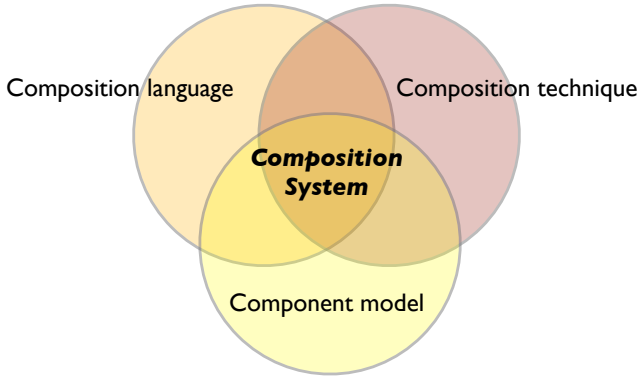


FIGURE 3.1: *A composition system consists of a* composition language, composition technique *and a* component model.

main instrument for controlling and restricting compositions. The component model of an ISC composition system aimed for Java could, for example, state that it is only possible to define method and class boxes (controlling how components may look). Furthermore, the component model could dictate that only method names may be variable (controlling how components may be transformed, defining their interfaces). The exact restrictions posed by a component model often differs between composition systems, depending on their precise requirements. The composition technique details how reusable units are joined together. The composition technique of ISC essentially constitutes integrating source code fragments (boxes) into other boxes via source code transformations, using their interfaces. That is, replacing boxes' variation points with other boxes (or fragments) used as values. The actual transformation of the code is performed by the provided basic composition operators *bind()* and *extend()*. The composition technique of ISC—and the same holds for GBM—is very general in the sense that it can be reused for many different composition systems. Figure 3.2 illustrates the main composition system parts for a generic composition system based on ISC.

Decomposition of composition systems Here we describe two previously discussed composition systems and see how they can be described according to the three parts: component model, composition technique and composition language. The Mjølner fragment system is described in Table 3.1, while a previously discussed composition system based on *RL* is described in Table 3.2.

Comparing ISC with grammar-based modularization

As ISC and GBM are conceptually closely related techniques, it is worthwhile highlighting their differences and similarities. Unavoidably, certain comments relate to the main demonstrator systems of the techniques, COMPOST and the Mjølner system, respectively.

- *Terminology.* The units of deployment—virtually the same: source code fragments of some underlying language—are in GBM called *fragment-forms*, and in ISC *fragment boxes*. Furthermore, explicit variation points are called *slots* in GBM and *hooks* in ISC.

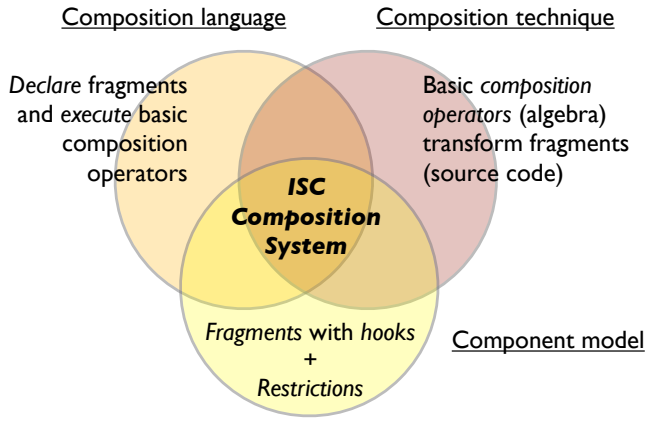


FIGURE 3.2: *A composition system for ISC has special requirements for the three parts of a general composition system (cf. Figure 3.1).*

The Mjølner fragment system

- *Component model.* The component model is similar to ISC's in the sense that fragments are considered as modules. The Mjølner fragment system imposes several restrictions in its component model: slots may only appear in certain places in fragments and be of certain carefully selected syntactic categories (*ObjectDescriptor*, *Attributes*, *DoPart*, and *MainPart* [29, p. 213]). As a consequence, only fragment-forms of the same selected syntactic categories may be defined as modules. Furthermore, only slots in fragment-forms may be transformed during composition, while the other parts are fully encapsulated. Also, the component model requires that the syntactic categories of slot applications and slot declarations are the same.
- *Composition technique.* The composition technique could be described as unifying slot declarations with slot applications via matching of names, and upon successful matching replacing the slots with the corresponding fragment-forms.
- *Composition language.* The composition language provides constructs for defining fragment-forms (seen using a graphical syntax in, for example, (2.5)). The binding of fragment-forms with slots is partly done using the *origin* construct, and partly specified implicitly based on the use of similar names for fragment-forms and slots. Some other constructs are provided for more detailed control of the composition process, but they are not further discussed here.

TABLE 3.1: *Dissection of the Mjølner fragment system.*

Reuse grammar from Example 2.2 (p. 39)
– *Component model.* The component model is formalized by the reuse grammar family $\mathcal{G}'$. It describes how fragments may look, and the *slot* construct ($\langle$*slot'*$\rangle$ in the grammar) explains where the fragment interfaces may appear, and hence how fragments may be accessed. Furthermore, $\mathcal{G}'$ constrains compositions since any intermediate composition results have to be well-formed sentences wrt. $\mathcal{G}'$.
– *Composition technique.* The composition technique is essentially the same as for the Mjølner fragment system: Slots are replaced by fragments treated as values.
– *Composition language.* The composition language could be the generic fragment language FL^{ABS} from Section 2.2.2.

TABLE 3.2: *Dissection of a reuse-grammar–based composition system for* RL.

A further note on terminology: In Chapter 2 we used terms such as 'fragment language' and 'fragment system.' We did this to stay close to the terminology of the Mjølner system. In this chapter however, we will use the terms 'composition language' and 'composition system,' to stay close to the terminology of ISC.

– *Explicit variation points.* Slots are in GBM programmed using a special slot construct, while ISC (as realized in COMPOST) uses naming conventions on existing (thus, overloaded) constructs of the underlying language for marking hooks. These are however merely design choices made by the demonstrator systems and could be different.

– *Implicit variation points.* ISC allows for implicit access of fragment boxes. This is not possible in GBM which solely considers explicit slots. As implicit variation points—by definition—are not marked by programmers, the permitted implicit access points are in ISC dictated by a component model specialized for the addressed underlying language.

– *Static vs. dynamic.* GBM, as implemented in the Mjølner fragment system, allows for separate compilation of fragments. Thus, fragment-forms can in principle be bound at run-time. Since COMPOST/J overloads language constructs for marking slots, certain box types can also be compiled in COMPOST/J (those coinciding with Java's compilation units, e.g. class boxes). However, ISC does not provide means for binding fragments at run-time, or by composing at the level of Java byte-code. Hence, ISC does not provide for separate compilation of fragment boxes such that the composition semantics is maintained.

– *Parameterization and extension.* Both ISC and GBM support fragment parameterization using hooks and slots, respectively. ISC supports extension explicitly via its *extend()* composition operator. Although less obvious, GBM also supports extension. If allowed by the underlying language, a single slot can be extended with several fragment-forms (e.g. if the slot appears in a statement list, cf. Example 2.2). Once the slot is removed, however, that particular location cannot

further be transformed. In ISC this is however possible using the fragment box's implicit interface.

3.2 Universal Invasive Software Composition

In contrast to other composition techniques, ISC allows for more flexible component adaptations by supporting both explicit and implicit component interfaces. Aspect-oriented programming with ASPECTJ [52], for example, is based on purely implicit interfaces. The Mjølner fragment system, as we have discussed, only supports explicit interfaces [63]. It is worth noting that an implicit interface only means that the reuse context is not declared, and should not be made equivalent to unrestricted component (fragment) access and transformation.

The main drawback of the COMPOST realization of ISC is its manually specified component models. By this we here mean the manual programming of each supported fragment box type, and the manual declaration and specification of how they may be transformed. Manual specification of a component model in with way is a tedious task and makes it very cumbersome to develop composition systems for various languages. So, while ISC is a general technique applicable to many different languages, the main question is how to realize this genericity in a manageable fashion. That is, how can the adaptation to ISC for a given language be automated?

The root cause as to why COMPOST requires manual specification of component models is that COMPOST is not *grammar-driven*. To address this problem—and allow for ISC to be more widely usable—we will leverage the grammar-based and automated adaptation techniques presented in Chapter 2 and extend it to also cover ISC. We will discuss these issues in the following section.

3.2.1 Grammar adaptation for ISC

The goal in this section is to extend the formalization of the GBM concepts from Chapter 2 to also accommodate ISC. As the main divergence from GBM is ISC's use of implicit interfaces, we need to adapt and extend our approach accordingly. This goal is illustrated in Figure 3.3 where a reuse grammar can be adapted to an ISC-*reuse grammar*. The adaptation step constitutes "annotating" the reuse grammar with restrictions for how fragments of the reuse grammar can be transformed implicitly. Then, instead of only requiring a simple fragment language (cf. Figure 2.1) we will discuss an extended language also able to access implicit interfaces, subject to restrictions as specified by the grammar annotations.

We do not need to extend the reuse grammar with additional syntactical constructs since implicit interfaces are never—by definition—marked in fragments. We would however like to be able to restrict how fragments can be transformed implicitly, as specified by composition programs. From a composition system perspective this can

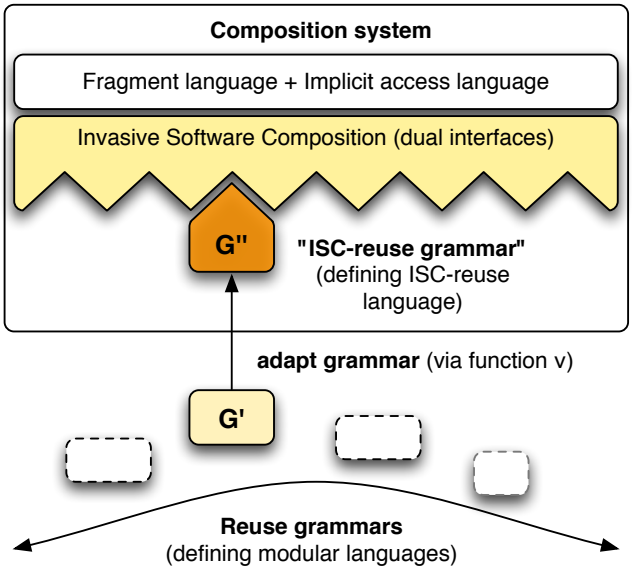


FIGURE 3.3: *Grammars adapted for GBM can be further adapted to ISC by also considering implicit interfaces. Such an adapted grammar is called an* ISC-reuse grammar *wrt. the reuse grammar.*

essentially be done in two ways:

1. *Preemptive*. Provide programmers (end-users) with a predefined set of constructs they may use to access fragments implicitly. Access of points in fragments not supported by such constructs is prevented, since means for doing so are not available.

2. *Non-preemptive*. Allow programmers (end-users) to access all locations in fragments—in *principle*—but declare restrictions that will be enforced on such accesses.

The COMPOST/J system follows the first alternative. For example, COMPOST/J provides the `<scope>.methodEntry` reference for implicitly accessing method entries in Java class boxes, where `<scope>` is a particular Java class and method name. Such a reference is just a declared name that is understood by the implementation of the composition system, somewhere linked to a manually written procedure that knows how to access such points in Java code. Other names exist for other implicit points (`<scope>.imports`, `<scope>.superClass` etc.). All other points in fragments are inaccessible in an implicit manner. By providing programmers with such a vocabulary, they have a language for talking and reasoning about the software entities in a meaningful way. Such capabilities are extremely useful to programmers, and are possible to provide when tailoring a composition system for a particular language. Again, this vocabulary simultaneously restricts programmers in what they can do, which is also the purpose.

In a generic approach, not making assumptions of the underlying language, such useful vocabularies cannot be predefined (there might not be any methods, hence no

⟨*vpoint'*⟩ ::= ⟨*slot'*⟩

TABLE 3.3: *Construct for general variation points in fragments.*

⟨*vpoint'*⟩ ::= ⟨*slot'*⟩
⟨*slot'*⟩ ::= « ⟨*ident'*⟩ »
⟨*ident'*⟩ ::= STRING

TABLE 3.4: *The* ISC*-grammar.*

method entries). But without a meaningful vocabulary, we cannot know how to restrict implicit fragment access. Leaving it entirely unrestricted—for example by allowing programmers to freely traverse fragments' abstract syntax trees, or similar—is not a viable solution since it leads to uncontrolled composition.

As a result, we will follow a middle way between the two approaches. We will not presume the kind of implicit interfaces that are desirable for fragments of different languages, and hence not predefine any such restrictions. We will, however, allow developers of composition systems to dictate the kind of implicit interfaces that should be possible to specify on fragments for a particular system. This will be done by "annotating" the grammars that the fragments being composed conform to. The grammar annotations serve two different purposes, arguably equally important:

1. *Conceptual.* By annotating grammars as a means for restricting implicit fragment access we can formalize and explain the restrictions by referring to the formalism of context-free grammars.

2. *Practical.* For practical purposes we can make use of such grammar annotations for generating constructs for *users* of composition systems. The set of generated constructs appropriately limits the users in accessing fragments implicitly, in a similar way as is done in COMPOST/J.

In the following we discuss the conceptual part, while the practical part is addressed in Section 3.2.3 and Section 3.3.

For the conceptual part we build upon our work from Chapter 2 where the extended base grammar—the reuse grammar—captures the component model of the base language, that is, describing how modules (fragments) look and are accessed.

We will in the following refer to implicit interfaces simply as (implicit) *variation points*. As a variation point is a rather vague notion, simply meaning a point in a fragment that is part of its composition interface, we could say that slots are kinds of variation points. We do this by augmenting the previously defined SLOT-grammar with the nonterminal ⟨*vpoint'*⟩ (assumed not to previously exist in any base grammar). Its definition can be found in Table 3.3. We call this extension of the SLOT-grammar for the ISC-grammar. The complete ISC-grammar is shown in Table 3.4.

To mark a certain base language construct as a valid variation point we transform the corresponding base grammar nonterminal via function: $\nu : (CFG, n) \rightarrow CFG$, where n is a nonterminal of the input CFG. To which nonterminals this function is

applied has to be specified by a developer. For a given input base grammar G, and nonterminal n, ν is defined by the following transformation steps, resulting in grammar G':

1. Union the ISC-grammar with G in the same way as for ψ (cf. Section 2.2.1).[4]
2. Introduce the new unit production rule: $\langle vpoint' \rangle ::= \langle n \rangle$.

Note that in G', after a single application of ν, nonterminal $\langle vpoint' \rangle$ generates the same set of strings as $\langle n \rangle$.

Theorem 3.1. *(*Safe grammar annotation*) Given CFG, say G, and nonterminal n defined in G, let* $\nu(G,n) = G'$. *Then G and* G' *generate the same languages, that is,* $L(G) = L(G')$.

Proof. We need to show that (i) any string generated by G can also be generated by G', and (ii) that G' does not generate any additional strings. To understand that (i) holds we simply need to recognize that no production rules from G are modified by ν. Hence the exact same derivation sequence used to derive a string l with G, $S \xrightarrow[G]{*} l$, can be used to derive l with G', $S \xrightarrow[G']{*} l$, where S is the start symbol of both G and G'. To convince ourselves that (ii) holds we acknowledge that the only additional derivation rule in G', compared to G, is the one unit production rule defining $\langle vpoint' \rangle$. Under the assumption that $\langle vpoint' \rangle$ is disjoint from the set of nonterminals N of G, it is not used by any of the original production rules from G, now in G'. Hence, starting from S we cannot derive any strings with G' that cannot be derived with G. □

Theorem 3.1 says that ν does not change the language generated by the input grammar. That is, ν is in a sense *safe*. This is the case since $\langle vpoint' \rangle$ is assumed not to exist in the input grammar G, and the start symbol of G is not changed in G'. This means that $\langle vpoint' \rangle$ is never used in the derivation of any string starting from the start symbol in G'. Hence, the introduction of the one unit production rule does not affect the generated language. So, what is really the point of its introduction? As mentioned, the intension of the grammar transformation function ν is to annotate grammars with information that can be used during composition to forbid certain fragment transformations. Thus, such annotations play a role for composition systems executing compositions, where users can be informed of forbidden, or unable to perform certain, fragment transformations.

While restricting compositions is important, in practical terms the grammar annotations do not necessarily have to result in grammar transformations. It is nonetheless important that the considered composition system "remembers" the specified restrictions (annotations) and enforces them during composition. The reason we introduce these fragment access restrictions as grammar annotations is that we can then explain the full ISC interface semantics wrt. the semantics of context-free grammars. Intuitively, we explain it like this: in a transformed grammar G', the nonterminal $\langle vpoint' \rangle$ generates a set of strings, and every string in this set, when appearing in a fragment valid wrt. G', can be considered part of the fragment's interface.

Example 3.1. Consider the rule language *RL*'s grammar from Example 2.1, call it G. Assume we want to say that *atom*s are accessible during composition as implicit

[4] Note that if this has already been done for the grammar (during some other step), it is not repeated.

variation points (that is, they are not marked in some special way in concrete fragments, but are still allowed to be modified during composition). So, we perform $\nu(G, atom) \rightarrow G'$. When composing fragments of G', the component model would allow us to use the implicit interfaces to perform the following transformation:

```
bonus(X,10) :- employee(X).   →   bonus(X,10) :-
                                  employee(X), friendly(X).
```
(3.1)

but not the following:

```
employee(john).   ↛   employee(john, 200).
```
(3.2)

The reason is that nonterminal $\langle num \rangle$ representing numbers is not "annotated" via the ISC-grammar in G', but nonterminal $\langle atom \rangle$ is. Hence, a user trying to perform the second transformation would get an error from the composition system, or simply be unable to do so.

■

We recall that a reuse grammar family $\mathcal{G}$ from Chapter 2 formally describes the component model for the language of the original base grammar. We want to extend this formal description of the component model when moving to ISC. That is, we want to integrate the grammar annotations restricting implicit fragment access with the notion of reuse grammars. We call such an augmented reuse grammar an ISC-*reuse grammar*.

Definition 3.1. *(*Context-free ISC-reuse grammar*) Let $\mathcal{G} = (N', \Sigma', P', \{s_1, \ldots, s_n\})$ be a reuse grammar, assumed to have used the* ISC*-grammar rather than the* SLOT*-grammar during its construction from the base grammar $G = (N, \Sigma, P, S)$. Let $I_{impl} \subseteq N$ be a set of nonterminals subject to annotations. Then we transform $\mathcal{G}$ by the following step:*

1. *For pairwise different $j_1, \ldots, j_m \in I_{impl}$, where $|I_{impl}| = m$, apply:*

$$\nu(\nu(\ldots\nu(\mathcal{G}, j_1)\ldots, j_{m-1}), j_m) = \mathcal{G}'$$

Then, the grammar $\mathcal{G}'$ is a context-free ISC-reuse grammar.

The ISC-reuse grammar $\mathcal{G}'$ captures and describes the full adaptation to ISC. It describes how fragments of the base language may look and how they may be accessed, both explicitly and implicitly. That is:

> If the ISC-reuse grammar $\mathcal{G}'$ is derived from base grammar G, then $\mathcal{G}'$ dictates the valid ISC components for $L(G)$ – both their valid structure and interfaces.

That is, a ISC-reuse grammar $\mathcal{G}' = (N, \Sigma, P, \{s_1, \ldots, s_n\})$ created from a base grammar G captures the component model for $L(G)$ and its composition system through the following observations:

1. The language $L(\mathcal{G}')$ contains all valid fragments of $L(G)$.

2. The nonterminal $vpoint' \in N$ of $\mathcal{G}'$ (coming from the ISC-grammar) generates all strings accessible in fragments of $L(G)$, both explicitly and implicitly. That is, if s is a string representing a fragment valid wrt. $\mathcal{G}'$, then a substring t of s is accessible during composition, if and only if:

$$t \in \{x \mid vpoint' \xrightarrow[\mathcal{G}']{*} x\}$$

Composing abstract syntax trees (ASTs) In Chapter 2 we were not very detailed on how fragments concretely were transformed, only assuming that strings representing fragments could be transformed by substituting substrings representing slots. ISC, however, traditionally does not manipulate strings, but composes fragments by transforming their abstract syntax trees (ASTs). We will in the following also assume that all composition takes place on fragments' ASTs. Remember that the Mjølner system [63] supports separate compilation of fragments with slots, and hence handle slots in a particular way. In contrast, we support slot constructs by introducing them into the relevant grammars. This allows us to parse fragments—including slots—into ASTs. This can be done for any grammar. Also, parser generation tools such as ANTLR can generate parsers that construct ASTs as the result of parsing fragments. When transforming slots in fragments, we transform nodes in ASTs that correspond to slots. When we transform implicit variation points, we transform "normal" AST nodes, that is, non-slot nodes. We call such a non-slot node for a *reference node*.

The types for slots was defined in Definition 2.6 (p. 47). The types for implicit variation points are slightly different, and in particular depend on the reference node and the chosen set I_{impl} representing implicit variation points.

Definition 3.2. *(*Grammatical types for implicit variation points*) Let G' be a* ISC*-reuse grammar derived from base grammar $G = (N, \Sigma, P, S)$ using the set $I_{impl} \subseteq N$ for representing valid implicit variation points. Let F be a fragment valid wrt. G' and IVP a reference node in the AST representing F. Then, the subtree with IVP as root node represents a substring s of F. Let $\tau^+(IVP)$ represent the set:*

$$\tau^+(IVP) = \{n \in N \mid \forall f \in L(n) : (F \xrightarrow{(s,f)} F') \in L(G')\}$$

The type of IVP can then be defined as follows:

$$\tau(IVP) = \tau^+(IVP) \cap I_{impl}$$

Notice that the definition of $\tau^+(IVP)$ above is similar to the defined notion of types for slots (cf. Definition 2.6 (p. 47)). The above definition intuitively says the following. The type of an implicit variation point IVP is the intersection of two sets:

1. *Grammar specific.* The set I_{impl} chosen as valid implicit interfaces for the entire ISC-reuse grammar.

2. *Fragment specific.* The types of the reference node. This is defined as the set of nonterminals $N' \subseteq N$ such that for each $n \in N'$:

$$\forall f \in L(n) : (F \xrightarrow{(IVP,f)} F') \in L(G')$$

Notice that Definition 3.2 ensures that the resultant fragment after an implicit composition step is valid wrt. the considered ISC-reuse grammar. Furthermore that the safety condition relates to the fragment being transformed and its grammar, rather than the precise structure of the fragment's AST.

Example 3.2. (*Implicit variation point safety*) Let's build on Example 2.7, where the fragment in (3.3) was composed.

```
bonus(X, 200) :- employee(X).
```
(3.3)

Assume that *atom*s are annotated to be variation points in the ISC-reuse grammar G'—hence $I_{impl} = \{atom\}$—and that we want to transform the fragment from (3.3) using its implicit interface. First remember that ISC composes fragments by transforming their abstract syntax trees (ASTs). Also remember that, in contrast to explicit slots, implicit variation points are not defined by names (we will discuss this in Section 3.2.3). Let the fragment F' in (3.4) be the same one as in (3.3), but where we, for the sake of this example, have made the node in the fragment's AST we want to transform explicit, and refer to it using the name *condition*.

```
bonus(X, 200) :- employee(X).
```
condition
(3.4)

The types for the implicit point *condition* is $\tau(condition) = \{atom, head, body\} \cap I_{impl} = \{atom\}$. Suppose we want to extend the rule body via its implicit interface (here: *condition*) with the G'-valid P-program g: `"efficient(X)"`. The types of g are $\{atom, head, body\}$. Since $\tau(g) \cap \tau(condition) \neq \emptyset$, this composition step is safe. The result F'' is shown in (3.5).

```
bonus(X, 200) :- employee(X), efficient(X).
```
(3.5)

In contrast, assume we want to extend the body list of (3.4) with the (assumed) G'-valid P-program `"X"`, call it g', whose types are $\tau(g') = \{term, var\}$. Since $\tau(g') \cap \tau(condition) = \emptyset$, the composition step is unsafe, and hence invalid.

■

Related sets of nonterminals for component models A final note on constructing reuse grammars. In Chapter 2, when constructing reuse grammars, we chose a subset $N_{slot} \subseteq N$, and a subset $N_{frgmt} \subseteq N$, wrt. a base grammar $G = (N, \Sigma, P, S)$, as desirable "slotable" constructs and valid fragment types, respectively. In this chapter, when constructing ISC-reuse grammars, we chose a subset $I_{impl} \subseteq N$ to represent accessible implicit variation points. All these sets are naturally closely related. As an example, consider the grammar of *RL*. If we defined $N_{slot} = \{atom\}$, then it seems natural that we should also have $atom \in N_{frgmt}$: If we want to bind *atom*s to slots, we must be able to define *atom*s. Likewise, if we want to transform *var*s implicitly, for example to rename a variable, we would have $var \in I_{impl}$, and also require $var \in N_{frgmt}$. In general, for the construction of a ISC-reuse grammar, we would have:

$$N_{slot} \cup I_{impl} = N_{frgmt} \tag{3.6}$$

The constraint in (3.6) is however only an intuitive and useful guideline, and not strictly enforced. We could also, for example, define $N_{slot} = \{term\}, I_{impl} = \emptyset$ and $N_{frgmt} = \{var\}$, which does not conform to (3.6). This would mean that we can declare

Interface	Operator	Usage	Comment
Explicit	*bind()*	Parameterization	*Slot application*
Implicit	*bind()*	Parameterization	*Replaces sub-fragments*
	extend()	Extension	*Extends list constructs*

TABLE 3.5: *The ISC composition operators and their usages.*

slots where *term*s may appear, but we can only define fragments of syntactic category $\langle var \rangle$. Note, however, that we will mainly rely on the constraint in (3.6).

3.2.2 Aligned composition algebra

When defining ISC in a grammar-driven fashion, and as an extension of GBM, it is useful to align the concepts and terminologies of their composition algebras. As can be seen in Example 3.2, ISC can transform fragments implicitly, as long as there is a way to reference the points in fragments that the composition algebra should operate on. It is the task of the composition language to support means of addressing implicit variation points in fragments. We will discuss this in Section 3.2.3. In this section we instead study exactly how the composition algebra of ISC goes beyond the one of GBM.

We recall that GBM supports parameterization of fragments through explicitly declared slots. Less obvious, but important to realize, is that GBM also supports extension. If the value fragment to be bound to a slot consists of a collection of fragments with valid types, then the target fragment is effectively extended. Hence, GBM supports both parameterization and extension, but only via explicit interfaces. In addition, each time an explicit interface is used, it is "consumed" and cannot be used again.

Example 3.3. (*Extension using slots*) Consider the *RL* fragment below with a slot in the rule body:

```
bonus(X, 200) :- employee(X), « condition ».        (3.7)
```

Then the slot *condition* can be bound by the fragment set:

```
overtime(X,Y)
gt(Y,50)                                            (3.8)
```

to produce the fragment:

```
bonus(X, 200) :- employee(X), overtime(X,Y), gt(Y,50).   (3.9)
```

■

The intuitive semantics behind slots is that they declare points in fragments that should be replaced, once, during composition. And this can be done using *bind()*, regardless if it is for simple parameterization or fragment extension. But in contrast to GBM, ISC has an explicit notion of extension via the *extend()* operator. However, as seen, there is never any need for the *extend()* operator to work on explicit interfaces.[5]

[5]If not for being able to extend fragments while leaving the slot operated on in place (as opposed to *bind()* which would remove it). But, as we regard this to be opposed to the understood semantics of slots, we disallow it. If a slot is desired to be left behind after a transformation, the value fragment may introduce a new slot.

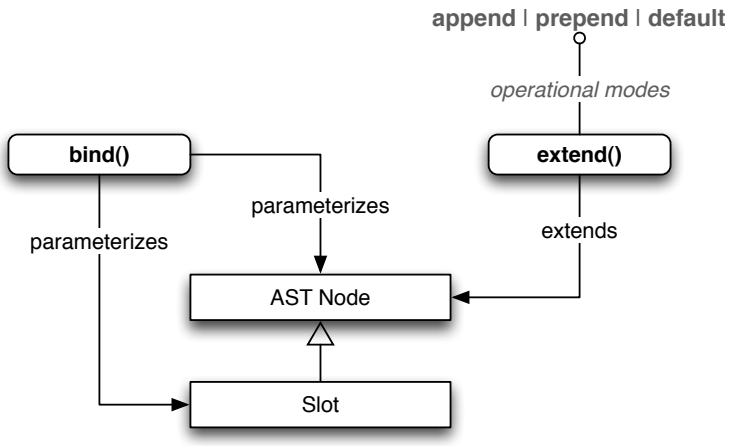


FIGURE 3.4: *The operator* bind() *can work on both explicit and implicit interfaces for simple parameterization or fragment extension. The* extend() *operator only works on implicit interfaces for the purpose of extension.*

By recognizing that parameterization via *bind()* can be used for fragment extension, we shall assign a very particular meaning to ISC's extension operator: true extension – *extend()* never replaces parts of fragments, but only always extends them. This means that *extend()* will only work on nodes of fragments' ASTs. But *bind()* can also be used on fragments' AST nodes, hence, working directly on their implicit interfaces. In this case, rather than replacing a slot in a fragment, a node in the fragment's AST is replaced. Such a replaced node in a fragment AST represents a *sub-fragment*. Again, this can be used both for simple parameterization and for fragment extension. Table 3.5 summarizes the ISC composition operators and their usages. By 'parameterization' in Table 3.5 we mean the replacement of something in the target fragment (a slot or an AST node), and by 'extension' we mean that nothing is replaced, but the target fragment is only extended.

Example 3.4. (*Using* bind() *on implicit interfaces*) Consider the fragment below, where we use the name *condition* for the node in the fragment's AST corresponding to the second atom in the rule body:

$$\texttt{bonus(X, 200) :- employee(X),} \underbrace{\texttt{efficient(X)}}_{condition}\texttt{.} \tag{3.10}$$

Then we can transform the fragment using *bind()* is a similar way as was done in Example 3.3 (cf. (3.9)).

■

The only contribution of the *extend()* operator is to allow for extension of list-like constructs using implicit interfaces. That is, when we do not want to replace anything in the target fragment, but *only* extend it. These different possibilities for the *bind()* and *extend()* operators, that are at the heart of the ISC algebra, are visualized in Figure 3.4. As can be noticed in Figure 3.4, there are three different *operational modes* that *extend()* can operate in: *append*, *prepend* and *default*. The reason we need

them is the following. Both *bind()* and *extend()*, when working on implicit interfaces, operate wrt. some reference node in the considered fragment's AST. For this reason, when applying *extend()* on such a reference node—which must be a node in a list-structure—it is not clear if the value fragment should be inserted before or after the reference node. That is, if the value fragment should be appended or prepended. The operational modes *append* and *prepend* control this. The *default* operational mode tries to be "smart"; it prepends if the reference node is the first node in the list-structure, and appends when the reference node is the last node. If the reference node is neither first, nor last, then the value fragment is appended (in the *default* mode).

Example 3.5. (*Extension using* extend() *on implicit interfaces*) Consider the fragment below, where we use the name *condition* for the reference node in the fragment's AST corresponding to the second atom in the rule body:

```
bonus(X, 200) :- employee(X), efficient(X) .        (3.11)
                              condition
```

Let the following be composition statements in pseudo-code using the *extend()* operator, operating in *prepend*, *append* and *default* mode, respectively:

```
prepend condition with "friendly(X)"        (3.12)
 append condition with "friendly(X)"        (3.13)
 extend condition with "friendly(X)"        (3.14)
```

Then each of the above composition statements would result in each of the following, respectively:

```
bonus(X, 200) :- employee(X), friendly(X), efficient(X).   (3.15)
bonus(X, 200) :- employee(X), efficient(X), friendly(X).   (3.16)
bonus(X, 200) :- employee(X), efficient(X), friendly(X).   (3.17)
```

If the first atom in the rule body from (3.11) was selected as reference node instead of the second atom, then the composition statement in (3.14) (using the *default* extension mode) would result in:

```
bonus(X, 200) :- friendly(X), employee(X), efficient(X).   (3.18)
```

This is because the *default* mode prepends when the first node in a list is used as the reference node.

■

Recall that reference nodes are always required in fragments' ASTs for implicit transformations. This causes a technical problem for empty list structures, for example, an empty method parameter list. In this case a reference node does not exist. This can however be solved in a concrete implementation by automatically generating special 'empty' reference nodes.

3.2.3 Generic composition language for ISC

In this section we discuss the requirements for a generic composition language for ISC. Notice that we used the term 'fragment language' in Chapter 2, while we here use the more general term 'composition language' (in line with the terminology of ISC). By 'generic' we here mean a composition language that can be used regardless of the underlying component language. A *component language* is the language in which fragments are written (that is, a language generated by an ISC-reuse grammar). In Section 2.2.2 we discussed the requirements for a generic fragment language for GBM called FL^{ABS} (only dealing with explicit interfaces). The goal here is to extend those requirements and deliver an approach for realizing a generic composition language also able to handle the implicit interfaces of ISC.

A good place to start is understanding what the composition language is in COMPOST/J. First recall that the component language in COMPOST/J is Java. Composition programs in COMPOST/J are also written in Java. So, does this mean that Java is the composition language in COMPOST/J? Nierstrasz explains in [66] that a composition language "supports the technical requirements of a component-oriented development approach by shifting emphasis from programming and inheritance of classes to specification and composition of components." [66, p. 147] This means that the true "composition language" in COMPOST/J is not really Java itself, but rather the Java types and methods that allow for talking about, and composing, Java fragments. We recall Java types such as `ClassBox`, and methods such as `createClassBox()`. So, Java is more of a surrounding environment—a platform—where the true composition language in form of COMPOST-provided types and methods reside. This distinction is important to recognize, and we make it explicit with the two definitions below.

Definition 3.3. *(*Core composition language*) We call the composition language which provides the vocabulary for talking about the software entities being composed, and how they should be composed, for the* core composition language*.*

The core composition language can then be different from the language which *enables* the core composition language:

Definition 3.4. *(*Host composition language*) We call any composition language which facilitates a core composition language for a* host composition language*.*

The host language can provide useful constructs that are not essential to the core composition language, for example, control-flow mechanisms, exception handling and reuse abstractions (e.g. procedures). A core composition language and a host composition language can also be the same. This is the case if the core composition language constructs are first-class entities of the host composition language. This is the case in, for example, ASPECTJ [52]. However, in other cases, which is true for COMPOST/J, the two are clearly different. In COMPOST/J, the host composition language is Java, while the core composition language is a Java-based API for defining Java boxes and for talking about their interfaces (how this API is used can be seen in Listing 3.5).

Based on this understanding, we now define two main properties of a generic composition language for ISC:

LR1 *Relation between core composition languages and component models.* The core composition language for COMPOST/J is tightly integrated with the component model (illustrated in Figure 3.5). It predefines means for declaring the supported fragment types, knowledge that typically resides in the component model. For

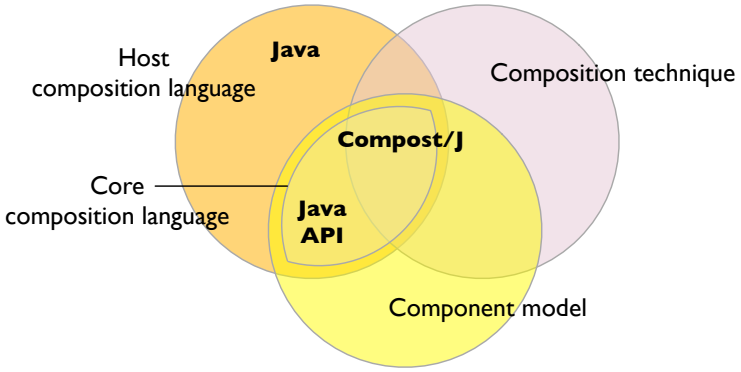


FIGURE 3.5: COMPOST/J *uses a Java-based API as the core composition language and has a tight connection with its component model.*

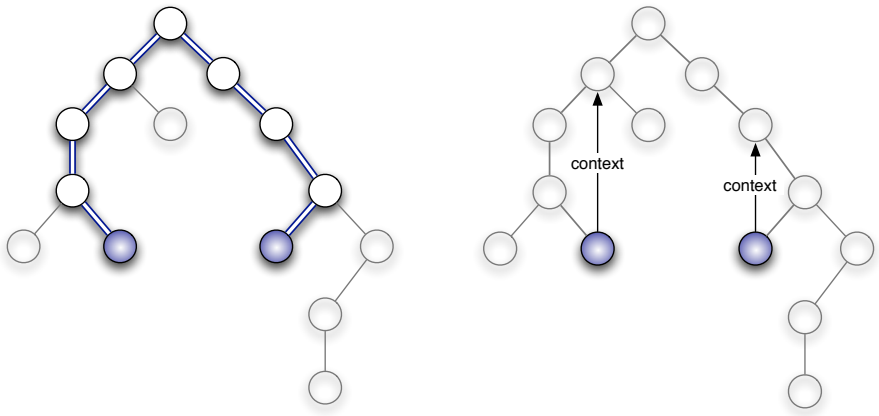
example, the method `createClassBox(String)` exists for declaring Java class boxes. This is not bad in itself, the problem is rather connected to the fact that COMPOST/J's core composition language—its provided API for working with Java boxes—is fixed and manually specified. For our framework, there are essentially two options for a useable and generic composition language:

1. The host composition language natively provides possibilities to work with fragments and realizes the required ISC composition algebra, or
2. The host composition language is any general-purpose language, while the core composition language is generated in accordance with the specified component model for the particular component language.

Neither of the above is the case in COMPOST/J. First, Java does not natively support working with our notion of fragments. Second, the COMPOST-provided API for working with Java fragments—the core composition language—is manually specified. So, support for a different component language must be manually specified in a similar way as for Java.

The first option is arguably not realistic for ISC since no such language exists, and to define one is not an easy task. The second option is more feasible since many different general-purpose languages exist that can be used as host languages. What is required is a way to generate the core composition language, the required API that can be used together with the host composition language for the purpose of composing fragments. Such a generative approach could be said to realize a generic core composition language: The generation is parameterized by any given component language (actually, its grammar).

LR2 *Accessing implicit interfaces.* If there exists no predefined core composition language, there also does not exist any predefined means of accessing implicit variation points in fragments. Remember that (explicit) slots are accessed via name references, but implicit variation points are by definition not named. A core composition language often provides convenient mechanisms to access certain points in fragments of the component language. For example, ASPECTJ provides simple means of accessing the entry and exit points of Java methods



(a) Explicit traversal with implicit context.

(b) Implicit traversal with explicit context.

FIGURE 3.6: *Two different traversal techniques of ASTs: explicit and implicit.*

(while the programmer has to specify exactly which methods should be considered). Without such predefined access points, the fragment programmers must have a better understanding of the component language structure (its grammar). Programmers must essentially traverse the abstract syntax trees (ASTs) of the fragments themselves, since no other means of access is provided for them. Predefined access points are not only attractive because of their convenience, but also because they effectively restrict how fragments may be accessed. That is, by assuming that there are no other means to access fragments implicitly. So, a generic composition language needs to provide the means to (at least partially) traverse the defined fragments' ASTs, subject to restrictions posed by the relevant component model.

In the following we focus on different ways of traversing fragments' ASTs. To select specific nodes in an AST, there are, generally speaking, two main approaches:

1. *Explicit traversal – implicit context.* The first possibility is to explicitly specify the path in the AST to reach the desired node. This is illustrated in Figure 3.6a where two nodes are selected via two different paths. Notice that by explicitly specifying the path to take to reach a particular node, the context that the node appears in is also known (implicitly). For example, consider the following *RL* rule:

   ```
   head(X) :- body(X).
   ```
   (3.19)

   If the selected nodes in Figure 3.6a correspond to the two variables (named `X`) in (3.19), then each of the expressions that select the nodes reveal if the selected node corresponds to the variable in the rule head, or the rule body.

2. *Implicit traversal – explicit context.* Another possibility is to simply express what *kind* of nodes should be selected in the AST, without caring so much about the details of *how* they are reached. This is illustrated in Figure 3.6b where the same nodes are selected as in Figure 3.6a, but without

```
<prgm>
  <stmt>
    <rule>
      <head>
        <atom>
          <predname>bonus</predname>
          <term><var>X</var></term>
          <term><num>200</num></term>
        </atom>
      </head>
      <body>
        <atom>
          <predname>employee</predname>
          <term><var>X</var></term>
        </atom>
      </body>
    </rule>
  </stmt>
</prgm>
```

LISTING 3.7: *XML representation of the fragment in* (3.20) *with highlighted text for successful matching as specified in Listing 3.8, and italicizes text for matching as specified in Listing 3.9.*

giving explicit path expressions to those nodes. That is, the traversal of the AST is implicit. But in this case—since no explicit path expressions were given to reach the nodes—it is not immediately clear in which contexts the selected nodes appear. If we use the same example as above and assume that we have selected the two variables from (3.19), then we must explicitly query the context for each node to find out if we have selected the variable in the rule head or in the rule body (see Figure 3.6b).

One of the tasks of the composition language in a generic ISC setting is to provide appropriate constructs for selecting nodes in fragments' ASTs. Furthermore, the composition programs written in the composition language must adhere to any given component model that restricts the access to the fragments' ASTs. Once the required (and permitted) AST nodes are selected, the ISC composition algebra can operate on those points to transform the fragments. In principle any suitable query language could be deployed to select the needed AST nodes.

3.2.3.1 Using Java as host composition language

Concretely, we will take an approach similar to the one used in COMPOST/J, but which is more general. That is, we will use Java as the host composition language. As in COMPOST/J, the core composition language will be a Java library, a Java API. This is convenient since it makes it possible to vary the core composition language, while the host composition language remains the same. Being able to vary the core composition language is a basic requirement of our framework approach, where different composition systems work with different kinds of software entities, and hence require the appropriate terms for reasoning about them.

However, in contrast to COMPOST/J, we do not provide a predefined set of APIs

```
1  public void compositionProgram()
2  {
3         IPrgm program = ... ;
4         program.accept(new RlVisitor() {
5
6                 public boolean visit(IAtom node) {
7                         // transform node
8                         ...
9                         return true;
10                }
11        });
12 }
```

LISTING 3.8: *Composition program selecting all* atoms *in a* RL *program.*

for particular base languages, for example, Java. We do not commit to any component language, and instead support a language-inclusive approach by being grammar-driven. This means that we generate the required Java APIs—the core composition languages—based on certain specifications: component model specifications. By allowing component model specifications to dictate the core composition languages, we effectively restrict the composition programs written in those languages, which is one of the points of the component models in the first place. Notice that providing a core composition language as a Java API is similar to providing a DSL for composition for a particular component language. In this view, the core composition language is a DSL embedded in Java (the host language).

How component models are specified and how the required Java APIs are generated is covered in Section 3.3. Here we instead focus on how the core composition languages, once they are generated, can be used for selecting the appropriate nodes in fragments' ASTs, such that the ISC composition algebra can operate on them. That is, we focus on how implicit interfaces can be managed in a generic framework lacking predefined, and language-specific, fragment access operators.

It should be noted that our technical realization is partially a consequence of choosing Java as the host composition language. A different host composition language would most likely result in a different solution, realized based on the constructs provided in that language.

We shall in the following, for demonstration purposes, use the well-known XML serialization format to encode ASTs. The *RL* fragment in (3.20) can be represented by an AST, for example by the XML-based tree-structure in Listing 3.7, where each tag label (name enclosed by < and >) corresponds to a nonterminal of the *RL* language grammar (cf. Example 2.1).

```
bonus(X, 200) :- employee(X).
```
(3.20)

In trying to make the selection of implicit variation points as easy as possible for developers, we will follow the second AST traversal possibility discussed above. That is, we assume an implicit traversal of ASTs and allow developers to specify certain criterion on which AST nodes should be selected. We illustrate with an example in Listing 3.8.

The Java types `IPrgm` and `IAtom` used in Listing 3.8 belong to the core composition language (assumed to have been generated), and allow us to work with *RL* programs and atoms. The object `program` is assumed to contain the program in (3.20). To im-

plicitly traverse the fragment's AST (Listing 3.7) we pass a "visitor" object to the `accept(Visitor)` method on the `program` object. The `accept(Visitor)` method will automatically iterate through the AST of the fragment on which it is called. For each AST node which is allowed to be transformed, it will invoke a `visit`-method on the received visitor object, passing the visited sub-fragment as an argument (with the visited AST node as root node). This gives the programmer an opportunity to transform the fragment implicitly. The default and generated `RlVisitor` class provides some empty `visit`-methods that can be overridden to do something useful. They all have the following signature:

```
public boolean visit([Type] node)
```

There exists one such method for each type (`[Type]`) that is supported by the underlying component model. If `false` is returned from the `visit`-method, the implicit traversal is aborted, but if `true` is returned, it is continued. Notice that there is a connection here to our previously introduced notion of "grammar annotations" in ISC-reuse grammars. We can override `visit`-methods for types that correspond to annotated nonterminals in the assumed ISC-reuse grammar. This is because the annotations indicate that such points in fragments may be transformed implicitly, and hence should be selectable. In the example in Listing 3.8, we assume that the types `IPrgm` and `IAtom` are supported by the component model (corresponding to *RL*'s nonterminals $\langle prgm \rangle$ and $\langle atom \rangle$). This means that we can override the method with the specific signature:

```
public boolean visit(IAtom node)
```

The overridden method in Listing 3.8 will be invoked for the nodes in the AST as indicated in Listing 3.7 (bold text indicates matching). That is, all the AST nodes that correspond to atoms are selected. To avoid having to define a new class where the appropriate methods are overridden, we have in Listing 3.8 made use of Java's ability to define *anonymous instances*, or *classes*. The code between Lines 4–11 in Listing 3.8 automatically creates, and instantiates, a new class that extends `RlVisitor` and contains the defined methods (here: `visit(IAtom)`). Again, since we here override `visit(IAtom)`, we have a chance to transform *atom*s within the main fragment (represented by the `program` object).

However, as mentioned, in such an implicit traversal approach, it can be necessary to explicitly query for the context of the selected nodes. For example, we might want to distinguish between the atom in the rule head and the atom in the rule body. This can be achieved as demonstrated by the program in Listing 3.9. Here we query for the context of the selected node using the `inContextOf([NodeType])` method. The method takes an AST node type as argument, which is here provided by the `RlUtil` class. The query in Listing 3.9 effectively checks if the selected atom is contained in the rule body. By querying for the context we can here avoid transforming the atom in the rule head. This refined selection is shown in Listing 3.7, where the bold and italicized text indicates selection.

We prefer the implicit fragment traversal approach over the explicit traversal approach since:

1. It relieves the programmer of some details of the underlying language; focus can be directed to the interesting points from a composition perspective, instead of having to formulate full AST path expression.

```
public void compositionProgram()
{
       IPrgm program = ... ;
       program.accept(new RlVisitor() {

              public boolean visit(IAtom node) {
                     if (node.inContextOf(RlUtil.BODY)) {
                            // transform node
                            ...
                            return true;
                     }
              }
       });
}
```

LISTING 3.9: *Composition program selecting all* atoms *in bodies of rules in a* RL *program.*

2. The approach is more declarative in nature, and hence arguably easier to understand.

3. There is a clear connection between overridden `visit`-methods and grammar annotations; there is a clear connection between access of implicit points and the specified component model.

3.2.3.2 Querying contexts

The following methods are currently supported for querying contexts of nodes:

– `<node>.inContextOf([NodeType])`

  Checks if the node operated on is in context of the node type passed as an argument.

– `<node>.isFirst()`

  Checks if the node operated on is the first member of a list.

– `<node>.isLast()`

  Checks if the node operated on is the last member of a list.

The `isFirst()` and `isLast()` context queries can be used to find out if, for example, a selected RL *atom* is the first or last atom in a rule body.

3.2.3.3 Composition algebra and FL^{ABS} support in core composition languages

To actually transform fragments, a core composition language must support the underlying ISC composition algebra. That is, the composition algebra detailed in Section 3.2.2. For our Java-based solution this means that the methods detailed in Table 3.6 should be provided for objects that correspond to valid fragments. The *bind()* and *extend()* (in its three modes) operators transform by *value*. This means that the value fragments being bound or extended are copies. We also provide the helper operator *collect()* that can be used to extend a "collector" fragment by *reference*. This means that if the collected fragments are transformed, this also affects the fragments

Operator	Parameters	Comment
Explicit interface		
`<frgmt>.bind(s,f)`	Slot name `s`, Value fragment `f`	Binds `f` to slots named `s` in `frgmt`
Implicit interface		
`<frgmt>.bind(f)`	Value fragment `f`	Replaces `frgmt` fragment with `f`
`<frgmt>.extend(f)`	Value fragment `f`	Extends `frgmt` with `f`, using `frgmt` as reference node (*default* mode)
`<frgmt>.prepend(f)`	Value fragment `f`	Extends `frgmt` with `f`, using `frgmt` as reference node (*prepend* mode)
`<frgmt>.append(f)`	Value fragment `f`	Extends `frgmt` with `f`, using `frgmt` as reference node (*append* mode)
Helpers		
`<frgmt>.accept(v)`	Visitor `v`	Applies visitor `v` on `frgmt`
`<frgmt>.collect(f)`	Fragment `f`	Extends collector fragment `frgmt` with `f`
`<frgmt>.print(file)`	File `file`	Prints content of `<frgmt>` to `file`

TABLE 3.6: *The main methods available on generated core composition language objects representing fragments.*

from which they were collected. We will see how this collector operator is used in Section 3.5 (Listing 3.19).

Our generated core composition languages are supersets of the abstract fragment language FL^{ABS}. The basic FL^{ABS} constructs are supported in the following way:

1. *Fragment declaration.* The types provided by the core composition language API can be used to declare fragments (cf. `IPrgm` in Listing 3.8).
2. *Slot application.* The `<frgmt>.bind(s,f)` method enables slot applications.
3. *Result specification.* The `<frgmt>.print(file)` method enables result specifications.

3.2.3.4 Refining and enriching core composition languages

As mentioned, it can be very useful for programmers to have predefined ways of accessing often reoccurring positions in fragments. A good example in Java is method entry and exit points. Staying with our *RL* language, the first and last position of rule bodies could be useful to refer to by name, simplifying adding *preconditions* and *postconditions* to rules. While we, as discussed, cannot predefine such convenient ways of accessing certain points in fragments, our Java-based solution enables an easy way to define such names for particular composition systems (e.g. for a composition system supporting *RL*).

Example 3.6. Assume we want to add preconditions and postconditions to *RL* rules. In the first case this would involve prepending a condition atom to the considered rule body atom list, while in the second case the condition atom would be appended. This can be done in the particular places we want to accomplish this, but if we want to have the possibility of reusing such fragment accesses across composition programs, we can encapsulate it in a Java method. The Java method name then essentially becomes a

```
1  public class RLLibrary {
2        public static RlVisitor precondition(final IAtom condition) {
3              return new RlVisitor() {
4                    public boolean visit(IAtom node) {
5                          if (node.inContextOf(RlUtil.BODY) && node.isFirst()) {
6                                // prepend condition
7                                node.prepend(condition);
8                          }
9                          return true;
10                   }
11             };
12       }
13
14       public static RlVisitor postcondition(final IAtom condition) {
15             return new RlVisitor() {
16                   public boolean visit(IAtom node) {
17                         if (node.inContextOf(RlUtil.BODY) && node.isLast()) {
18                               // append condition
19                               node.append(condition);
20                         }
21                         return true;
22                   }
23             };
24       }
25 }
```

LISTING 3.10: *Reusable AST visitors for* RL *fragments. The first method accesses the first atom of rule bodies (to add a precondition), and the second the last atom (to add a postcondition).*

term that can be used to talk about such fragment locations. In Listing 3.10 we define two such methods encapsulating pre- and postcondition access to *RL* rules.

Such defined means of accessing *RL* fragments can be used as shown in Listing 3.11. The composition program in Listing 3.11 defines three *RL* fragments, a program `program` and two atoms, `precond` and `postcond`. The `program` fragment is implicitly transformed using the defined fragment access methods from Listing 3.10.

Let us assume that the fragment `program` corresponds to the following fragment:

```
p(X) :- q(X), r(X).
```
(3.21)

Further, that `precond` and `postcond` correspond to `"pre(X)"` and `"post(X)"`, respectively. Then the `program` fragment would be the following after Listing 3.11 had executed:

```
p(X) :- pre(X), q(X), r(X), post(X).
```
(3.22)

■

Other valuable fragment access points can be defined in a similar way as in Example 3.6, and for completely different component languages.

The possibility to define commonly used fragment access methods for specific composition systems in reusable libraries essentially allows to *refine* and *enrich* core composition languages using the standard generic fragment access approach (cf. Listing 3.10). Notice that the implementation of the enriched fragment interfaces—the

```
1  public void compositionProgram()
2  {
3          IPrgm program = ... ;
4          IAtom precond = ... ;
5          IAtom postcond = ... ;
6
7          program.accept(RLLibrary.precondition(precond));
8          program.accept(RLLibrary.postcondition(postcond));
9          ...
10 }
```

LISTING 3.11: *Composition program using predefined* names *defined in Listing 3.10.*

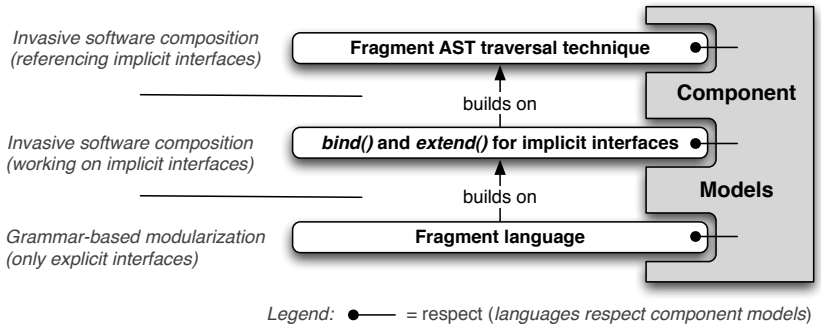


FIGURE 3.7: *Composition language hierarchy, from GBM to ISC.*

defined and reusable methods, e.g. `precondition()` in Listing 3.10—still have to respect the underlying component model. It is merely a means to achieve, in a generic framework, the predefined and convenient access methods often provided by specialized composition systems such as COMPOST/J or ASPECTJ. A basic requirement for all of this is naturally that the host composition language supports a notion of procedures or methods, which is often the case for general-purpose languages.

Summary

We here briefly summarize what was achieved above wrt. composition languages (see Figure 3.7). We started from an abstract representation of a basic fragment language used in grammar-based modularization approaches—FL^{ABS}. To also support the implicit interfaces of ISC we extended this basic language with the two additional operators *bind()* and *extend()* that work on implicit interfaces, corresponding to the missing parts of ISC's basic composition algebra. Regardless of the sophistication of the particular composition language, the component models restrict how compositions can be defined.

The primitiveness of ISC's composition algebra is due to not considering how transformational points in fragments are actually accessed. The access of *explicit* variation points is not the concern of the programmer since names are used to refer to such points. The main challenge is to support ISC's *implicit* interfaces in a generic framework. What is needed is controlled transformation of fragment ASTs. We achieved this by providing a visitor-like pattern as a means to reference implicit points, but where the

usage of such patterns are controlled by the underlying component model. More examples of how composition systems can be specified and used are given in Section 3.3. Using a generative approach we can realize a two-faced composition language: Java is used as the host composition language, while core composition languages, specific to composition systems, are generated as Java APIs.

Notice that our solution is based on using Java as the host composition language, and that this choice dictates the solution. Using a different host composition language can possibly result in a different, perhaps better, solution. A language such as Scala[6] supports visitor patterns as a primitive construct and the solution could possibly have been even simpler if Scala was used as the host composition language. This is, however, not further investigated here.

3.3 Developing U-ISC–based composition systems

In this section we describe more practical details of how to construct composition systems based on the already discussed notions. In Section 3.3.1 we provide a concrete language, called C_mSL, for specifying the fragment-based component models of component languages. Later we will give examples of its use.

3.3.1 Component model specification language (C_mSL)

The constructs of the language presented here directly correspond to the requirements of component model specification. That is, it provides the necessary constructs for dictating the structure of a reuse grammar $\mathcal{G}$, starting from a base grammar G.

– **Base grammar declaration.** First we need to declare which base grammar is being considered. This is done using the `extends` construct:

  ⟨*decl*⟩ ::= `extends` ⟨*url*⟩ `@` ⟨*ident*⟩ `as` ⟨*url*⟩ .

  where the ⟨*url*⟩ nonterminal represents a URL and nonterminal ⟨*ident*⟩ a string. The first URL is used to specify the file containing the base grammar. The identification string gives us a name to refer to the grammar in subsequent statements. Finally, the second URL specifies the location where the transformed grammar should be stored.

– **Slot declaration.** It should be possible to specify nonterminals in a grammar that should have corresponding slot constructs. For this purpose we use the `slotify` construct:

  ⟨*slotify*⟩ ::= `slotify` ⟨*nonterminal*⟩ .

  where ⟨*nonterminal*⟩ represents a nonterminal in the base grammar. The `slotify` construct corresponds to the ψ function as discussed in Section 2.2.1. We also recall that there is an alternative function ψ^* that creates type-specific slot constructs. Hence, we provide this function as a construct in our language as well:

  ⟨*slotify**⟩ ::= `slotify*` ⟨*nonterminal*⟩ .

[6] `http://www.scala-lang.org/`

⟨*cmsl*⟩	::=	⟨*decl*⟩ ⟨*stmt*⟩* ⟨*fragtypes*⟩
⟨*stmt*⟩	::=	⟨*slotify*⟩ | ⟨*slotify**⟩ | ⟨*comment*⟩

TABLE 3.7: *The main part of the* C_mSL*-grammar.*

– **Fragment types.** We also want to be able to dictate which fragment types are definable according to the component model. This is done using the `fragtypes` construct, using the following syntax:

⟨*fragtypes*⟩ ::= `fragtypes {` ⟨*nonterminal*⟩ (`,` ⟨*nonterminal*⟩)* `}`

where each nonterminal belongs to the base grammar. This construct does not affect or transform the base grammar, but is rather used by the framework implementation to restrict fragment definitions.

Every nonterminal in the above is assumed to be preceded by an identifier followed by a dot (`.`). The identifier represents the base grammar as specified using the `extends` construct. This is only an implementation convention, see Listing 3.12 for an example (p. 88). The main part of the *CmSL* grammar is show in Table 3.7, using the above defined constructs. Notice that a *CmSL* specification is not required to specify any slots.

As discussed, there is, in general, a difference between fragment types that are allowed to be defined, and fragment types that may be referenced implicitly during composition. In Chapter 2 we used a subset N_{frgmt} of the nonterminals of the base grammar for the former, and in this chapter "annotations" for the latter. Here, however, we use the `fragtypes` construct for both these purposes. That is, here we assume $N_{frgmt} = I_{impl}$. The reason for this is related to the implementation, where we use the same core language Java types (e.g. `IAtom`), both for defining fragments (N_{frgmt}) and for accessing fragments implicitly (I_{impl}). This is only a limitation in the current realization, and not a conceptual limitation. The real limitation in this case lies in the fact that one might want to allow definition of certain fragment types (e.g. to define fragment values for slots), but not allow implicit transformation of the same fragment types. That is, one would perhaps like to have, for some fragment type n: $n \in N_{slot}$, $n \in N_{frgmt}$ and $n \notin I_{impl}$, which currently is not possible.

3.3.2 Component model generation from *CmSL* specifications

Based on a *CmSL* specification, a component model can be generated. Here we cover the essentials of what is generated. First, the required slot constructs are injected, according to ψ, into the appropriate places in the base grammar, as dictated by the `slotify` constructs.[7] Second, appropriate core composition language types are generated, based on what is specified using the `fragtypes` construct. The generated types have the following name convention: `I + [Typename]`. Assume that the following (partial) *CmSL* specification is given, where `file:rl.gr` is the location of the *RL*

[7] In the remainder we will not use the ψ^* construct, but only the ψ construct.

grammar:

```
extends file:rl.gr @ rl as file:rlx.gr
...                                                    (3.23)
fragtypes { x.rule, x.atom }
```

Then the two types `IRule` and `IAtom` will be generated. In fact, these types are Java interfaces (hence the `I`). Their implementation classes are also be generated: `IRuleImpl` and `IAtomImpl` (perhaps unconventionally, the `I` stays in the name). The generated interfaces (types) subclass the generic interface `IFragment`, that defines methods corresponding to the ISC composition algebra from Table 3.6. The implementation types, named `I + [Typename] + Impl`, additionally implement a static *load(String)* method. This method can be used to load fragments. The *load(String)* method can be used in the following way:

```
IRule stmt = IRuleImpl.load("p(X):- q(X).")
```

As an aside, a fragment type (e.g. `IRule`) can not only represent a single fragment, but also a collection of fragments of the same type. A visitor class is generated that automatically can traverse fragments' ASTs. The naming convention of this class is: `[BaseLanguageName] + Visitor`. For our rule language *RL*, this class would hence be named: `RlVisitor`. It should be mentioned that all generated interface types (fragment types) are equipped with an *accept([Visitor])* method with a suitable signature. For *RL*, the fragment types would have a method with signature *accept(RlVisitor)*. The grammar specific visitor class implements a set of methods that can be overridden. One method is generated for each fragment type specified using `fragtypes`, and each method look as follows:

```
public boolean visit(I[Typename] node){ return true; }
```

A `visit`-method returning `true` means that the visitation of the fragment AST should continue after the particular visit. Returning `false` aborts the visitation.

Figure 3.8 exemplifies the set of generated classifiers for *RL* based on $N_{frgmt} = \{rule, atom\}$. The classifiers generated based on N_{frgmt} are shaded in grey. Two additional comments: (i) The generated type interfaces (e.g. `IAtom`) also demands a copy method for copying the content of fragments during composition. (ii) The visitor class (e.g. `RlVisitor`) also provides a constructor taking a fragment as an argument (not depicted in Figure 3.8). The fragment passed as the parameter to the constructor can be accessed using the method `getParamFragment()`. This constructor can be used for different purposes. How these constructs are used is demonstrated in Section 3.5. The class `RlAlgebra` in Figure 3.8 is generated to contain (Java) type information that is specific to the particular component language (in this case *RL*). The generic class `Algebra` implements the necessary composition algebra (promised by `IFragment`).

3.4 Tooling – REUSEWARE/AIR

In this section we give an overview of the involved tooling for developing the described composition systems. The techniques and ideas that have been described have been prototyped and implemented on top of the REUSEWARE Composition Framework

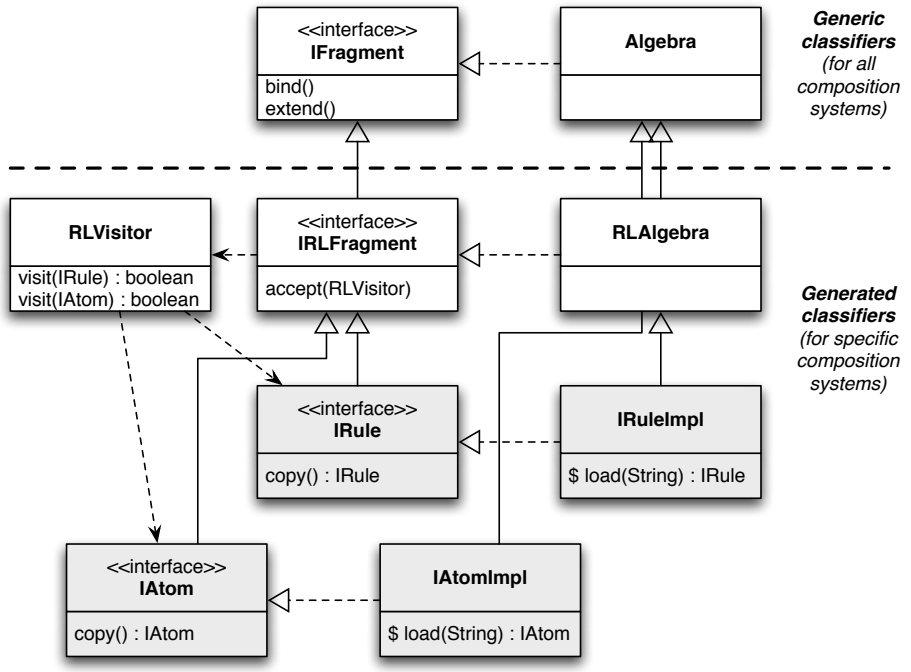


FIGURE 3.8: *An illustration of the generated classifier hierarchy for a core composition language in a composition system. Classifiers in grey are specific to what fragment types are desired (here* rule *and* atom *from* RL*). The classifiers above the dotted line are reused for each composition system, while those below are generated specifically for a certain component language (here* RL*).*

(or simply REUSEWARE) developed in the Software Technology group at the Technical University of Dresden [49].[8] REUSEWARE in turn is built on top of the Eclipse Modeling Framework (EMF).[9] EMF includes a modeling language (called Ecore) that can be used for describing languages as Ecore models. REUSEWARE provides functionality for transforming a simple form of language grammar specification into Ecore models. Having languages specified by Ecore models is helpful because all the tooling provided by EMF can be used to manipulate instantiations of such models. A valid instantiation of a language model is a program. These language models only describe the abstract syntax of languages. REUSEWARE also provides means to describe the concrete syntax of the languages by referring to these abstract syntax models. By employing the parser generator framework ANTLR[10], REUSEWARE can also generate parsers for the specified languages. These components of REUSEWARE are illustrated on the lower parts of Figure 3.9 (there called REUSEWARE/CORE), but are not further detailed here since they are not part of the contributions of this thesis (see [49] and `http://reuseware.org` for more details).

The implementation contribution of this thesis is called REUSEWARE/AIR and is

[8] `http://reuseware.org`
[9] `http://www.eclipse.org/emf/`
[10] `http://www.antlr.org/`

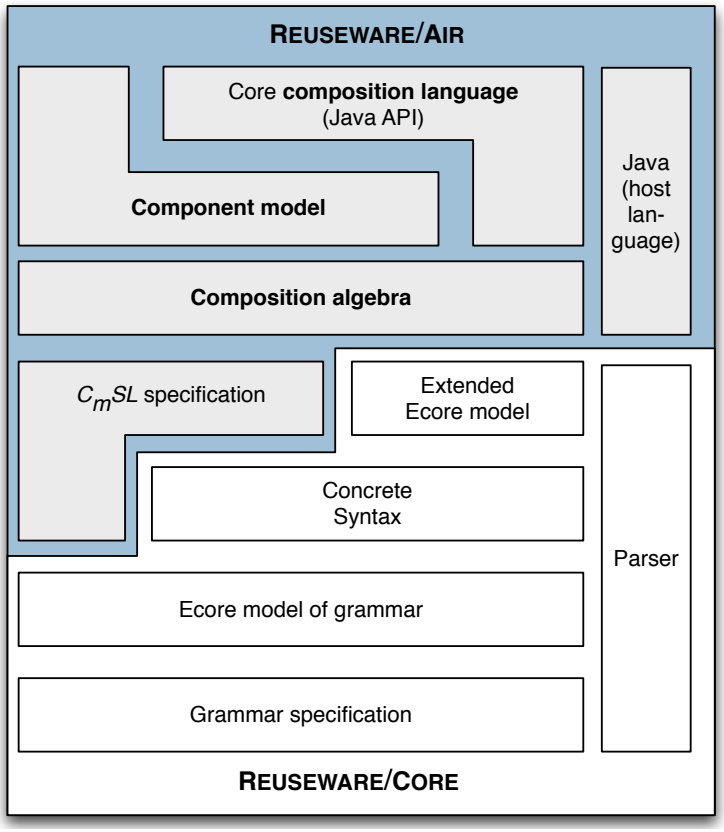


FIGURE 3.9: *Architectural overview of the* REUSEWARE *Composition Framework, including both* REUSEWARE/CORE *and* REUSEWARE/AIR. REUSEWARE/CORE *provides basic functionality for modeling languages, generating parsers and manipulating models (the Eclipse Modeling Framework is employed for this).* REUSEWARE/AIR *is developed on top of* REUSEWARE/CORE *and contains the prototypical implementation for the concepts discussed in this thesis.*

```
extends file:rl-grammar.gr @ rl as file:rlx-grammar.gr .

% 1) slot constructs

% 1a) atoms should be slotable
slotify rl.atom .

% 1b) we want a type specific slot construct for numerals
slotify* rl.num .

% 2) component model restrictions: fragment types
fragtypes { rl.prgm, rl.stmt, rl.rule, rl.atom, rl.num, rl.predname }
```

LISTING 3.12: CmSL *program specifying a component model for rule language* RL.

implemented on top of the REUSEWARE Composition Framework, hence makes use of it.[11] The components of this contribution is shown in the upper part of Figure 3.9. They are the following:

1. Specification of the *CmSL* language for describing component models (cf. Section 3.3.1). The interpretation of *CmSL* specifications has also been realized using the REUSEWARE framework (via bootstrapping techniques), but its description is out of the scope of this presentation.

2. Creation of extended language models (based on existing REUSEWARE techniques, but augmented with some technical details for the purpose of REUSEWARE/AIR).

3. Generation of core composition languages (cf. Section 3.3.2). That is, appropriate Java APIs are generated for working with fragments on the Java platform.

4. Implementation of required composition algebra. This involves both the primitive ISC algebra (based on the original implementation in REUSEWARE, but extended and adapted for the purpose of REUSEWARE/AIR), as well as the identified requirement of controlled implicit fragment transformations (cf. Section 3.2.3).

3.5 Examples: U-ISC–based composition systems

We here exemplify the development of two composition systems, one for *RL* and one for a simplified version of Java, here called Java^{-}.

3.5.1 Composition system for simple rule language

Here we will look at an example dealing with the *RL* language grammar (see Example 2.1, p. 27). We will use the *CmSL* language presented above to specify a tailored component model for *RL*. The *CmSL* program in Listing 3.12 defines two slot constructs and a list of fragment types that are considered valid (and hence also which sub-fragments that can be accessed implicitly). The result of this grammar extension can be found in Table 3.8. For this example we will refer to this grammar and its component model, as specified in Listing 3.12, as $\mathcal{G}(RL)$.

[11] The name "Air" is used in light of the desire to develop a lightweight composition framework.

⟨*prgm*⟩	::=	⟨*stmt*⟩*	⟨*atom*⟩	::=	⟨*atom'*⟩ | ⟨*slot'*⟩
⟨*stmt*⟩	::=	⟨*rule*⟩ | ⟨*fact*⟩	⟨*slot'*⟩	::=	« ⟨*ident'*⟩ (: ⟨*ident'*⟩)? »
⟨*rule*⟩	::=	⟨*head*⟩ :- ⟨*body*⟩ .	⟨*num*⟩	::=	⟨*num'*⟩ | ⟨*num-slot'*⟩
⟨*head*⟩	::=	⟨*atom*⟩	⟨*num-slot'*⟩	::=	« ⟨*ident'*⟩ : #num# »
⟨*fact*⟩	::=	⟨*atom*⟩ .	⟨*ident'*⟩	::=	STRING
⟨*body*⟩	::=	⟨*atom*⟩ (, ⟨*atom*⟩)*	⟨*predname*⟩	::=	STRING
⟨*atom'*⟩	::=	⟨*predname*⟩ (⟨*term*⟩ (, ⟨*term*⟩)*)	⟨*const*⟩	::=	STRING
			⟨*var*⟩	::=	CAP_STRING
⟨*term*⟩	::=	⟨*const*⟩ | ⟨*var*⟩ | ⟨*num*⟩	⟨*num'*⟩	::=	NUM_STRING

TABLE 3.8: *A* ISC*-reuse grammar for rule language* RL.

```
bonus(X, <<bonus : #num#>>) :-
       employee(X),
       <<bonuscondition>>.

reducepay(X, <<minus : #num#>>) :-
       employee(X),
       <<reducecondition>>.
```

LISTING 3.13: *Main paycheck program (*`file:main.rl`*).*

```
employeee(X) :-
     marketing_employee(X).
employee(X) :- sales_employee(X).

bonuseligible(X) :- hardworking(X).
bonuseligible(X) :- efficient(X).

reprimand(X) :- lazy(X).
reprimand(X) :- showsuplate(X).
```

LISTING 3.14: *Rules for aligning company terminology (*`file:align.rl`*).*

The valid fragment types as specified in Listing 3.12 do not appear in the concrete grammar in Table 3.8. This information is instead used for generating the appropriate core composition language. That is, Java types that we will use in our composition programs. The fragments in Listings 3.13–3.16 are all well-formed fragments wrt. $\mathcal{G}(RL)$, and describe a rule-based paycheck management system for a small fictitious company. Listing 3.13 contains some rules for deciding when employees receive bonuses or paycheck cuts. Since the company consists of several different and rather independent departments, Listing 3.14 provides rules for aligning the different terminologies used within the different departments. Listing 3.15 is a fact base detailing the sales department, while Listing 3.16 covers the marketing department.

```
sales_employee(steve).
sales_employee(marco).
hardworking(steve).
lazy(marco).
```

LISTING 3.15: *Database for sales department (*`file:sales.rl`*).*

```
marketing_employee(john).
marketing_employee(sarah).
showsuplate(john).
efficient(sarah).
```

LISTING 3.16: *Database for marketing department (*`file:marketing.rl`*).*

```
1  public void compositionProgram() {
2
3          // ------ load fragments ------
4          IPrgm program = IPrgmImpl.load("file:main.rl");
5          IPrgm align = IPrgmImpl.load("file:align.rl");
6          IPrgm sales = IPrgmImpl.load("file:sales.rl");
7          IPrgm marketing = IPrgmImpl.load("file:marketing.rl");
8
9          // ------ explicit interface (with inline fragment specification) ------
10         program.bind("bonuscondition", IAtomImpl.load("bonuseligiable(X)"));
11         program.bind("reducecondition", IAtomImpl.load("reprimand(X)"));
12         program.bind("bonus", INumImpl.load("200"));
13         program.bind("minus", INumImpl.load("100"));
14
15         // ------ implicit interface ------
16         program.extend(align);
17         program.extend(sales);
18         program.extend(marketing);
19
20         // ------ result specification ------
21         program.print("file:paycheckprogram.dl");
22 }
```

LISTING 3.17: *Composition program for a paycheck query program in* RL.

The different fragments can be maintained and modified independently of each other, and cover different concerns of the overall company structure. On payday the human resource (HR) department manager wants to find out where bonuses are due, and where pay cuts can be made. The program in Listing 3.17 accomplishes this task by composing the involved fragments into a usable query program. The final query program is constructed by binding the slots in Listing 3.13 with the appropriate values, as decided by the HR manager, and then extending the same fragment with the other fragments. The composition result can then be found in `file:paycheckprogram.dl`.

The composition result can be used to issue the following query: `bonus(X,Y)`. That is, querying who gets a bonus, and how much. The result would be:

$$\{X = steve,\ Y = 200\}, \{X = sarah,\ Y = 200\}$$

The query `reducepay(X,Y)` would give the following result:

$$\{X = marco,\ Y = 100\}, \{X = john,\ Y = 100\}$$

Notice that the composition program in Listing 3.17 adheres to the component model $\mathcal{G}(RL)$.

A simple example of how we could misuse the component model would be to write the composition program in Listing 3.18. First, $\mathcal{G}(RL)$ does not allow to define variables as fragments (Line 7, since nonterminal $\langle var \rangle$ was not mentioned during component model specification using the `fragtypes` construct). That is, the Java types `IVar` and `IVarImpl` are not generated and can therefore not be used. Second, even if we were allowed to define variables as fragments, the bind operation in Listing 3.18 is not valid since the type of any fragment bound to slot `value` must be of type *num*.

A slightly more complicated example of a correct composition program is given in Listing 3.19. There the `program` fragment is implicitly transformed by extending

```
1 public void compositionProgram() {
2
3       // ------ load fragments ------
4       IPrgm program = IPrgmImpl.load("file:main.rl");
5
6       // ------ bind fragments ------
7       program.bind("bonus", IVarImpl.load("X"));
8 }
```

LISTING 3.18: *Type-incorrect composition program.*

rules matching a specific condition with the additional body atom specified by the fragment `advice`. The condition is that the head predicate name is "bonus" (only one rule in our fragments matches this condition). To achieve this we first have to collect (extract) the rules that will be transformed. This is done between Lines 5–22. The collected rules can then be transformed, and this is done between Lines 24–33. This composition program in Listing 3.19 would transform the fragment `program` to start as in Listing 3.20 (notice that only the first rule is transformed implicitly).

Notice that in Listing 3.19 we first "collected" the rules we wanted to transform using a certain pattern:

1. Construct an empty fragment that is sent as a parameter to the constructor of the involved visitor class (here: `RlVisitor`). We call this empty fragment for the "collector fragment." If we want to nestle further into the original fragment (its AST), we can pass the collector fragment along by referring to it using the `getParamFragment()` method.

2. When the collector fragment is to be used, it can be retrieved within the anonymous instance using the `getParamFragment()` method.

3. Collect any sub-fragments of interest by using the method `collect()`.

Consider what would happen if we had not used `collect()`, but instead `extend()`. The "collection" would have worked the same, but transforming the collected fragments would leave the original fragment, from which they were collected, unmodified. So in that case the composition result would be the same for Listing 3.17 and Listing 3.19. When we want any transformations to affect the original source fragment we have to follow the above-described pattern. We shall find the need to use it again.

3.5.2 Composition system for Java$^-$

In this section we discuss applying our composition techniques to a subset of the Java language. This subset allows to specify some simple Java-like constructs, such as defining classes, methods and attributes (we will refer to self-explanatory language constructs such as "compilation unit," "method," "method name," "statement," "super type"). This language is presented for exemplary purposes and should only be considered as a toy language to demonstrate our composition technology. We call the language Java$^-$ (`javamm` or `jm` in code).

A *CmSL* component model specification can be found in Listing 3.21. It specifies that super types can be substituted by slots, and that we are able to define familiar Java-like constructs (as discussed above), and hence also be able to transform them

```
public void compositionProgram() {
	// ------- load and transform fragments (from Listing 3.17) -------
	...
	// ------- collect rules matching the specific condition -------
	IRule rules = new IRuleImpl();
	program.accept(new RlVisitor(rules) {
		public boolean visit(final IRule rule) {
			rule.accept(new RlVisitor(getParamFragment()) {
				public boolean visit(IPredname name) {
					// query context
					if (name.inContextOf(RlUtil.HEAD)) {
						// collect matching rules
						if (name.toString().equals("bonus")) {
							getParamFragment().collect(rule);
						}
					}
					return true;
				}
			}) ;
			return true;
		}
	});

	final IAtom advice = IAtomImpl.load("friendly(X)");
	rules.accept(new RlVisitor() {
		public boolean visit(IAtom node) {
			if (node.inContextOf(RlUtil.BODY) && node.isLast()) {
				// append condition
				node.append(advice);
			}
			return true;
		}
	});

	// ------- result specification -------
	program.print("file:paycheckprogram.dl");
}
```

LISTING 3.19: *Composition program using the implicit fragment interface to weave in an additional rule condition (represented by fragment* `advice`*) on rules matching a specific condition (having a particular head predicate).*

```
bonus(X, 200) :- employee(X), bonuseligiable(X), friendly(X).

reducepay(X, 100) :- employee(X), reprimand(X).

...
```

LISTING 3.20: *Result from composition program in Listing 3.19.*

```
extends file:javamm.gr @ jm as file:rjavamm.gr .

% 1) super types should be slotable
slotify java.Supertype .

% 2) component model restrictions: fragment types
fragtypes { jm.CompilationUnit, jm.Method, jm.Methodname,
            jm.Statement, jm.Parameter, jm.Modifier, jm.Supertype }
```

LISTING 3.21: CmSL *program specifying a component model for a simplified Java language Java*$^-$.

```
public compositionProgram() {

    // ------- load fragments -------
    ICompilationUnit cu = ICompilationUnitImpl.load("file:Machine.java");

    // ------- transform fragments -------
    // ------- explicit interface -------
    cu.bind("superType", ISupertypeImpl.load("ControlSystem"));
    // ------- implicit interface -------
    JavaFrgmtLibrary.methodExitLogWithMethodName(cu, "set\\w*", "Set:");

    JavaFrgmtLibrary.methodEntry(cu, JavaFrgmtLibrary.METHOD_VIS.PUBLIC,
           IStatementImpl.load("System.out.println(\"public method\")"));

    cu.print("file:result.java");
}
```

LISTING 3.24: *Composition program for Java*$^-$ *fragments. The program makes use of the Java*$^-$ *fragment composition library detailed in Listing 3.25 in Appendix 3.A.*

implicitly. We do not present the transformed grammar here, instead we focus on what we can do as a result of this specification.

With a generated composition system based on Listing 3.21 we can write, for example, the simple class in Listing 3.22. The class `Machine` defines a virtual "machine" holding a value (`value`). The value can be set and retrieved using defined public methods. The class also keeps track of the most recent value (`oldValue`). From outside the class, the old value can only be retrieved, and not set, since the setter (`setOldValue()`) is not declared to be public. The class also makes use of the possibility of leaving its super class unspecified using a slot named `superType`. Suppose we want to compose a new class based on this class. Say we would like to parameterize the class by binding the super class slot with some value, and weave in some debugging code: some print statements giving informing as to which methods are executed. This can be accomplished using the composition program in Listing 3.24. This composition program makes use of the generated core composition language resulting from the component model specification in Listing 3.21 (e.g. Java type `ICompilationUnit`). The composition program also makes use of a Java$^-$ fragment composition library that has been written using the same generated core composition language. The fragment composition library can be found in Listing 3.25 in Appendix 3.A (p. 98).

The composition program in Listing 3.24 does three main things:

```
1  public class Machine
2    extends <<superType>> {
3
4    private int value;
5    private int oldValue;
6
7    Machine() {
8      value = 1;
9    }
10
11   Machine(int v) {
12     value = v;
13   }
14
15   public void setValue(int v) {
16
17     setOldValue(value);
18     value = v;
19
20   }
21
22   public int getValue() {
23
24     return value;
25   }
26
27   void setOldValue() {
28     oldValue = value;
29
30   }
31
32   public int getOldValue() {
33
34     return oldValue;
35   }
36 }
```

LISTING 3.22: *Java⁻ class defining a machine with unspecified super class (`file:Machine.java`).*

```
1  public class Machine
2    extends ControlSystem {
3
4    private int value;
5    private int oldValue;
6
7    Machine() {
8      value = 1;
9    }
10
11   Machine(int v) {
12     value = v;
13   }
14
15   public void setValue(int v) {
16     System.out.println("public method");
17     setOldValue(value);
18     value = v;
19     System.out.println("Set:  setValue");
20   }
21
22   public int getValue() {
23     System.out.println("public method");
24     return value;
25   }
26
27   void setOldValue() {
28     oldValue = value;
29     System.out.println("Set:  setOldValue");
30   }
31
32   public int getOldValue() {
33     System.out.println("public method");
34     return oldValue;
35   }
36 }
```

LISTING 3.23: *Java⁻ class from Listing 3.22 with parameterized super class and logging code.*

1. It binds the super type slot `superType` with value "ControlSystem" (Line 8).

2. It adds a logging statement as the last statement in each method whose method name starts with "set" (Line 10). The output of such a log statement is the string "Set:" proceeded by the method name.

3. It adds a print statement as the first statement in each public method (Line 13).

Exactly how the involved class is transformed using the generated core composition language is captured in the reusable Java$^-$ fragment composition library (Listing 3.25 in Appendix 3.A). Notice the recurrence of the "collect fragment" pattern mentioned at the end of Section 3.5.1. The result of the composition can be seen in Listing 3.23.

This example has demonstrated how a rather simple specification (Listing 3.21) can give rise to flexible means of transforming fragments of the underlying component language (here Java$^-$). These transformations are done by relying on the underlying approach of invasive software composition, its composition algebra and controlled access of fragments (thanks to generation of a core composition language based on the component model specification).

This example has just demonstrated some few possible transformations, others are surely possible. It can be noted that this example covered two interesting and useful means of weaving (à la aspect-oriented programming) value fragments (advices) into a core fragment:

1. The transformation on Line 10 in Listing 3.24 selects methods based on their names (must be prefixed by the string "set"), and includes this information in the woven advice (here by including the method name in the log statement). That is, the advice is parameterized by the context where it is composed.

2. The transformation on Line 13 in Listing 3.24 uses a single non-modified advice (a print statement) in all locations in the core fragment (compilation unit `cu`) where the composition context matches (the method is declared to be public).

As can be noted in the Java$^-$ fragment composition library, the solution to the above compositions are solved a little differently. In the latter case, all the matching composition contexts can first be extracted using the "collect fragment" pattern. After this is done, the advice can be woven into the appropriate place. In the former case, however, all the matching composition contexts cannot be extracted as a first and separate step. This is because we need to use the context information for configuring the advice that is to be woven into the particular composition context. If we first collect, and then weave, the connection between the composition context and the advice is lost, or at least made more complicated. So, instead we deal with the advice immediately upon finding each matching composition context (cf. method `methodEntryExitLogWithMethodName/4` in Listing 3.25).

3.6 Summary

Let us recap what we accomplished in this chapter. Our goal was to extend the grammar-driven notions from Chapter 2 to also include the approach taken by ISC. That is, to go from specification of grammar-driven fragments systems to development of *grammar-driven invasive composition systems*. We addressed the following issues:

- *Grammar adaptation for ISC interfaces.* In contrast to GBM, ISC also considers implicit fragment interfaces. For this reason we extended the grammar adaptation from Chapter 2 with the function ν. Formally, the function ν is used to annotate nonterminals of base grammars to encode how fragments may implicitly be transformed. The resulting grammar is called an ISC-reuse grammar.

- *ISC component models.* These grammar adaptions are mainly discussed in the context of formally encoding the component models of the languages described by the grammars. Formally, the nonterminal ⟨*vpoint'*⟩ (from the included and general ISC-grammar) in a ISC-reuse grammar generates all strings that may be transformed during composition. In this way is the component model semantics encoded in the ISC-reuse grammars. That is, a ISC-reuse grammar captures, formally, how fragments may be composed.

- *Composition algebra: GBM vs. ISC.* It is important to understand the relationships between the composition algebra provided in fragment systems based on GBM, and the algebra in composition systems based on ISC. We clarified this relationship and distilled exactly how the ISC composition algebra goes beyond the one provided for GBM.

- *Composition languages.* The main difficulty for a generic and grammar-driven ISC approach lies with the composition language. In particular, due to ISC's implicit interfaces. Ideally, support for implicit fragment access should directly be provided by the used composition language. However, such a suitable and generic composition language does not exist. To address this problem we enabled a usable and practical approach by generating core composition languages for each composition system, embedded in a reusable host composition language. In practical terms, Java is used as the host composition language, while each generated core composition language is a Java library. The core composition languages come with a set of constructs that can be used to compose fragments, using both their explicit and implicit interfaces. Most importantly, the core composition languages support the ISC composition algebra.

- *Developing composition systems.* To enable the development of composition systems in practice we provided the component model specification language C_mSL. A C_mSL specification can generate a core composition language that supports the precise compositions as declared in the C_mSL specification. This means that the development of a component model dictates the possible composition programs that can be written using the corresponding (core) composition language. We demonstrated how composition systems can be developed and used by exemplifying one for the rule-based language *RL*, and one for a subset of Java called Java$^-$.

In the introduction to this chapter we mused that by universalizing ISC we will be in a better position to understand the limits and qualities of ISC as a composition approach. The following are some of our observations and experiences:

- Working with ISC's fragments and composition algebra is rather primitive and does not always feel like the ideal way of producing and building software. This is probably even more true in our grammar-driven approach compared to the language-tailored approach taken by COMPOST. The reason for this is that the

programmer (user of the composition system) needs to have a very good understanding of the grammar of the component language for accessing implicit variation points in fragments.

– We already mentioned that a generalized ISC approach will necessarily loose some of the benefits of the tailored solution provided by, for example, COMPOST/J. In COMPOST/J it is possible to constructs fragment boxes using a tailored Java-based API. This is demonstrated with the following example:

```
1 // define Java compilation unit box
2 CompilationUnitBox c;
3 JavaProgramFactory factory = c.getFactory();
4 // transform: introduce a package declaration
5 c.findHook("imports").
6       bind(factory.
7              createImport(factory.createPackageReference("java.io.File")));
```

Here we create a Java compilation unit and introduce a package import statement (using the ISC hook `"imports"`). In this case, Java's type system can verify part of the composition. For example, that the creation of the import statement works correctly (Line 7). It works correctly if `createPackageReference()` returns the type expected by `createImport()`. A similar approach to creating fragments is used in [12] and [94]. In our general approach we cannot create such detailed APIs, and hence not take advantage of Java's type system for validating the creation of fragments. To create the import statement fragment in our approach, we would do something like this:

```
1 IImportDeclaration decl =
2       IImportDeclarationImpl.load("import java.io.File");
3 ...
```

In this case, the construction of the fragment can only be validated when it is parsed. Hence, we do not provide fragment creation APIs. This is one of the benefits from COMPOST/J we give up to have a more general approach. However, from a practical viewpoint, this sacrifice does not seem very severe. Developers are unlikely to want to construct fragments using a API, rather than just specifying the fragments directly.

– What has helped when using our generated composition systems is the clarified understanding of the differences between GBM and ISC (which is based upon GBM) wrt. their composition algebras (cf. Section 3.2.2). This especially relates to understanding that extension can be achieved via the binding of slots.

– COMPOST provides predefined component models and fragment composition libraries. What is clear when generating composition systems is the added power given to developers: New languages can be addressed and developers are empowered by being able to develop and use their own fragment composition libraries (cf. Appendix 3.A). However, due to the grammar-driven nature of these composition systems, the composition libraries can be quite fragile. For example, if the grammar evolves or is modified, the composition system has to be regenerated which might affect the consistency of any developed composition libraries.

Even though working with U-ISC–based composition systems can seem primitive, we will make use of their flexibility and generality in the next chapter, and show how we can make their deployment easier for end-users.

3.A Appendices

In Section 3.A a fragment composition library for composing Java$^-$ fragments can be found. The code makes use of the core composition language (Java API) generated from the component models specification in Listing 3.21 (these Java classes have the namespace `org.reuseware.air.language.javamm`). A composition program making use of this library can be found in Listing 3.24.

Source code for Java$^-$ composition system

```
1  package javamm;
2
3  import java.util.regex.Pattern;
4  import java.util.regex.PatternSyntaxException;
5
6  import org.reuseware.air.language.javamm.ICompilationUnit;
7  import org.reuseware.air.language.javamm.IMethod;
8  import org.reuseware.air.language.javamm.IMethodname;
9  import org.reuseware.air.language.javamm.IModifier;
10 import org.reuseware.air.language.javamm.IStatement;
11 import org.reuseware.air.language.javamm.algebra.JavammVisitor;
12 import org.reuseware.air.language.javamm.impl.IMethodImpl;
13 import org.reuseware.air.language.javamm.impl.IStatementImpl;
14
15 public class JavaFrgmtLibrary {
16
17    private enum METHOD_POS { ENTRY, EXIT }
18    public enum METHOD_VIS { PUBLIC, PRIVATE, PROTECTED }
19
20    /**
21     * Method entry point cut
22     *
23     * @param cu
24     * @param methodName
25     * @param advice
26     */
27    public static void methodEntry(ICompilationUnit cu, String methodName,
          IStatement advice) {
28       methodEntryExit(cu, methodName, advice, METHOD_POS.ENTRY);
29    }
30
31    /**
32     * Method entry point cut
33     *
34     * @param cu
35     * @param methodName
36     * @param advice
37     */
38    public static void methodEntry(ICompilationUnit cu, METHOD_VIS visibility,
          IStatement advice) {
39       methodEntryExit(cu, visibility, advice, METHOD_POS.ENTRY);
40    }
41
42
```

```
43  /**
44   * Method exit point cut
45   *
46   * @param cu
47   * @param methodName
48   * @param advice
49   */
50  public static void methodExit(ICompilationUnit cu, String methodName,
        IStatement advice) {
51      methodEntryExit(cu, methodName, advice, METHOD_POS.EXIT);
52  }
53
54  /**
55   * Extract matching methods
56   *
57   * @param cu
58   * @param pattern
59   */
60  private static IMethod extractMatchingMethods(ICompilationUnit cu, final
        String pattern) {
61
62      // extract methods
63      IMethod matchingMethods = new IMethodImpl();
64      cu.accept(new JavammVisitor(matchingMethods) {
65
66          public boolean visit(final IMethod method) {
67              method.accept(new JavammVisitor(getParamFragment()) {
68
69                  public boolean visit(IMethodname mn) {
70
71                      // check for pattern
72                      if (matchName(pattern, mn.toString())) {
73                          // extract method
74                          getParamFragment().collect(method);
75                      }
76                      return true;
77                  }
78              });
79              return true;
80          }
81      });
82
83      return matchingMethods;
84  }
85
86  /**
87   * Extract public methods
88   *
89   * @param cu
90   * @param methodName
91   */
92  private static IMethod extractMethodsByVisibility(ICompilationUnit cu, final
        METHOD_VIS visibility) {
93
94      // extract methods
95      IMethod matchingMethods = new IMethodImpl();
96      cu.accept(new JavammVisitor(matchingMethods) {
97
98          public boolean visit(final IMethod method) {
99              method.accept(new JavammVisitor(getParamFragment()) {
100
101                 public boolean visit(IModifier mod) {
```

```
102
103                     if (visibility == METHOD_VIS.PUBLIC) {
104                         if (mod.toString().equals("public")) {
105                             // extract method
106                             getParamFragment().collect(method);
107                         }
108                     }
109                     return true;
110                 }
111             });
112             return true;
113         }
114     });
115
116     return matchingMethods;
117 }
118
119
120 /**
121  * Code for method entry and exit
122  *
123  * @param cu
124  * @param methodName
125  * @param advice
126  * @param mode
127  */
128 private static void
129 methodEntryExit(ICompilationUnit cu, final String methodName,
130         final IStatement advice, final METHOD_POS mode) {
131
132     // get methods to transform
133     IMethod matchingMethods =
134         extractMatchingMethods(cu, methodName);
135
136     // transform
137     methodEntryExitAdvice(matchingMethods, advice, mode);
138 }
139
140 /**
141  * Method entry point cut with method name message
142  *
143  * @param cu
144  * @param methodName
145  */
146 public static void methodEntryLogWithMethodName(ICompilationUnit cu, String
      methodName, String msg) {
147     methodEntryExitLogWithMethodName(cu, methodName, msg, METHOD_POS.ENTRY);
148 }
149
150 /**
151  * Method exit point cut with method name message
152  *
153  * @param cu
154  * @param methodName
155  */
156 public static void methodExitLogWithMethodName(ICompilationUnit cu, String
      methodName, String msg) {
157     methodEntryExitLogWithMethodName(cu, methodName, msg, METHOD_POS.EXIT);
158 }
159
160 /**
161  * Log message with method name parameterized (method entry/exit point cut)
```

```
162     *
163     * @param cu
164     * @param methodNamePattern
165     * @param advice
166     * @param mode
167     */
168    private static void
169    methodEntryExitLogWithMethodName(ICompilationUnit cu,
170           final String methodNamePattern, final String msg, final METHOD_POS
                mode) {
171
172        // extract methods
173        cu.accept(new JavammVisitor() {
174
175            public boolean visit(final IMethod method) {
176
177                method.accept(new JavammVisitor() {
178
179                    public boolean visit(IMethodname mn) {
180
181                        if (matchName(methodNamePattern, mn.toString())) {
182                            // construct advice
183                            final IStatement advice =
184                                IStatementImpl.load("System.out.println(\"" +
185                                        msg + " " +
186                                        mn.toString() + "\")");
187                            // transform method
188                            methodEntryExitAdvice(method, advice, mode);
189                        }
190                        return true;
191                    }
192                });
193                return true;
194            }
195        });
196
197    }
198
199    /**
200     * Code for method entry and exit
201     *
202     * @param cu
203     * @param methodName
204     * @param advice
205     * @param mode
206     */
207    private static void
208    methodEntryExit(ICompilationUnit cu, METHOD_VIS visibility,
209           final IStatement advice, final METHOD_POS mode) {
210
211        // get methods to transform
212        IMethod matchingMethods =
213            extractMethodsByVisibility(cu, visibility);
214
215        // transform
216        methodEntryExitAdvice(matchingMethods, advice, mode);
217    }
218
219    /**
220     * Advice methods, depending on the 'mode' parameter (ENTRY or EXIT)
221     *
222     * @param methods
```

```
223      * @param advice
224      * @param mode
225      */
226     private static void methodEntryExitAdvice(IMethod methods, final IStatement
             advice, final METHOD_POS mode) {
227         // transform
228         methods.accept(new JavammVisitor() {
229 
230             public boolean visit(IStatement stmt) {
231                 // methodEntry
232                 if (mode == METHOD_POS.ENTRY) {
233                     if (stmt.isFirst())
234                         stmt.prepend(advice);
235                 }
236                 // methodExit
237                 else if (mode == METHOD_POS.EXIT) {
238                     if (stmt.isLast())
239                         stmt.append(advice);
240                 }
241                 return true;
242             }
243         });
244     }
245 
246     /**
247      * Returns true if pattern matches name, false otherwise
248      *
249      * @param pattern
250      * @param name
251      * @return
252      */
253     private static boolean matchName(String pattern, String name) {
254 
255         try {
256             if (Pattern.matches(pattern, name))
257                 return true;
258         } catch (PatternSyntaxException e) {
259             System.err.println("Incorrect pattern '" +
260                     pattern + "': " + e.getMessage());
261         }
262         return false;
263     }
264 
265 }
```

LISTING 3.25: *Fragment composition library for composing Java⁻ fragments.*

4

Embedded Invasive Software Composition

In Chapters 2 and 3 we discussed existing generic modularization and composition techniques and showed how they can be universalized and applied to arbitrary formal languages. We demonstrated how (core) composition languages can be generated from component models specifications. These generated composition languages appropriately restricts the kind of components that may be defined, and effectively what kind of compositions that may be specified by programmers. In addition, the composition languages support the underlying ISC composition algebra. The composition algebra of ISC essentially boils down to the primitive composition operators *bind()* and *extend()*, corresponding to the general notions of software parameterization and extension, respectively. By supporting these two fundamental software composition techniques, and being based on principles that are language-agnostic, ISC is clearly a very general approach. A consequence of this generality is that the offered composition algebra can be nothing but primitive. The application of ISC's composition operators can be said to constitute *low-level composition steps*. For example, extending a Java class with a method box, or parameterizing a super-class. An example of this was given in Section 3.5.2 (p. 91). Describing compositions on this lower level can be cumbersome and uninviting to programmers, since such low-level steps often fail to capture any larger meaning wrt. the overall composition. The realization of many software abstractions require a set of low-level composition steps to be executed as a unit in a *high-level composition step*. As demonstrated, this can partly be accomplished by building up fragment composition libraries collecting high-level composition steps in methods or procedures. Even though this is convenient, the overall construction and design of the final result must still be done on the lower level; the developer has to think and work on the level of fragments and ISC's low-level operators. Aßmann has the following to

say about ISC's low-level operators:

> "[T]he basic operators are not expressive enough, since they are so general. [...] Software designers will not like designing with a minimal pattern language. Instead they will need languages with more domain-specific, tailored, and adequate composition operators. [...] And I believe that such languages will be the software construction languages of the future."
>
> *– Uwe Aßmann, Invasive Software Composition [5, p. 278]*

The "minimal pattern language" referred to here is essentially ISC's basic composition operators. A good questions to ask is what "domain-specific, tailored and adequate" means in this context. The following is our interpretation of these terms:

- **Domain-specific.** Ideally there should be composition operators available that have relevance to, and a connection with, the component language in question. Using the primitive *bind()* and *extend()* operators, for example, for both Java and a rule-based language (e.g. *RL*), is not optimal.[1]
- **Tailored.** Different component languages can require different notions of fragments. That is, they may need different component types or abstractions. The composition language and approach can be considered tailored if it supports components and compositions that are suitable for the considered component language.
- **Adequate.** When it comes to supporting different component types that relate to a particular component language, the usage and composition of those components should correlate to their expected semantics. That is, the composition operators need to be adequate enough to handle the expected usage of such components.

The main contribution of this chapter is a realization of the vision for ISC stated above. Thus, relieving software engineers of working with the "minimal pattern language," here equaled to ISC's fundamental composition algebra, and instead finding a way of providing them with more appropriate composition operators that, in summary:

1. Have a close connection to the underlying component language.
2. Satisfy the particular compositional needs, using the appropriate abstractions.

To achieve this we will leverage the stepwise ISC language adaptation from Chapters 2 and 3. A consequence of enabling the development of "domain-specific, tailored, and adequate composition operators" in general is that it will allow us to address an open problem in a quite general setting: the problem of how to extend DSLs with component-oriented constructs (new and useful abstractions). Addressing the problem based on the general composition capabilities of ISC gives us a broad platform to stand on, and does neither restrict languages to address, nor component types desired for different DSLs. Two applications that demonstrate this approach and provide such

[1] We prefer the term "domain-appropriate" over domain-specific, because we argue that we need composition operators that are not necessarily specific to the domain of a language, but rather *appropriate* for the domain. This because composition operators should support abstractions (components), but many useful abstractions are not necessarily domain-specific.

domain-specific and adequate composition operators are discussed in detail in Chapters 5 and 6. In this chapter we focus on the general motivation, methodology and connection to the composition technology of Chapters 2 and 3.

This chapter is structured as follows. In Section 4.1 we briefly discuss what we consider to be problematic with ISC, and hint at a solution. In Section 4.2 we discuss what domain-appropriateness means, in particular wrt. components and composition statements. In Section 4.3 we detail how we can connect the general ISC composition operators with more domain-appropriate and intuitive constructs. In Section 4.4 we explain what is needed to develop composition systems with this notion of domain-appropriateness. Section 4.5 presents an example of such a composition system. Finally, in Section 4.6, we summarize the achievements of the chapter, and discuss some remaining interesting issues.

4.1 Taming Invasive Software Composition

By looking at a composition program for ISC (e.g. Listing 3.5, p. 58), it is clear that ISC is a metaprogramming approach. This means that ISC inherently assumes the users of composition systems to consider software artifacts (programs and fragments) as data to be transformed and operated on. There is no possibility for programmers—advanced or beginners—of avoiding this explicit metaprogramming. Metaprogramming, while powerful, is generally considered to be a difficult technique and has a high learning curve. Using metaprogramming techniques, programmers do not only have to know about the problem domain in which their programs will run, or the (hopefully) intuitive language constructs used to write those programs, but they also have to clearly understand the underlying structure of the language they are programming in. Not that metaprogramming cannot be useful, but it is clear that it adds complexity.

In a sense, ISC abstracts from any particular underlying component language and treats them all very much the same. This is illustrated in Figure 4.1 by the *abstraction curve* up to the highest point. Even with a tailored core composition language, giving programmers the terms to define appropriate fragments, the natural connection to the component language is lost; ISC treats source code fragments and transforms them into desired results, all necessarily detailed by a programmer (by writing metaprograms). One of our objectives is to regain the closeness to the base language in which the composed fragments are written, and reduce the exposure to ISC's metaprogramming facilities. That is, in a sense to "tame" ISC by making it more usable, effective and attractive to end-users. Hence, allowing programmers (cf. Figure 4.1) to work with abstractions that are appropriate for the particular language in which the programmers write their code (this is illustrated by the downward part of the *abstraction curve* in Figure 4.1). If the language is a DSL, abstractions appropriate for the language also (should) make them appropriate for the domain.

While ISC traditionally exposes every user to its "bare bones," we will separate between the *developer* of a composition system, and the *user* of the same (cf. Figure 4.1). The developer has to understand the underpinnings of ISC, while the user can be restricted to only take advantage of the services provided by developed composition systems. Hence, the generality and flexibility of ISC can be exploited in the construction of composition systems, while the end-user experience is greatly simplified. The need for the general approach provided by ISC is not lessened by this separation of user roles, since a plethora of languages are still needed to be addressed, all of which should be catered for. Hence, the generality of ISC is needed on the *developer level*. However,

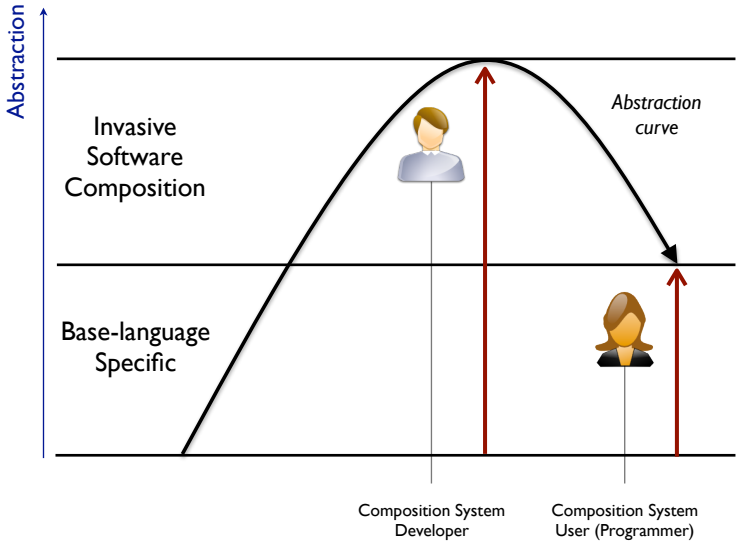


FIGURE 4.1: *It is important to distinguish two abstraction levels. One where primitive ISC concepts are handled directly by a composition system* developer, *and one where* programmers *can work with more domain-appropriate concepts and constructs.*

solving particular software engineering problems, like component-based development for DSLs, allow us to restrict how ISC is applied on the *user level*.

4.2 Domain appropriateness

A crucial requirement for developing and specifying successful abstractions is domain-appropriateness. That is, for an abstraction to be useful for, and adopted by, programmers, they must feel that the abstraction is related to their programming or development activities. Fragments in the way we have defined them certainly are abstractions since they can be configured for new reuse contexts. However, since the notion of fragments is universal and very general—not domain-specific—that abstraction is far from being domain-appropriate. Neither in the way fragments are defined, nor how they are composed using the available basic (low-level) composition operators.

We believe that programmers and developers ultimately do not want to design with fragments as first-class entities. They rather want to develop with classes, libraries, procedures, modules or packages etc. That is, they want to use notions and concepts that are related to their work task, and that fit their vocabulary. Both BETA and COMPOST essentially force developers to work directly with fragments and does not explicitly provide any other form of (reuse) abstraction. However, it should be clear that all these mentioned abstraction forms can to a large degree be *viewed* as fragments, that is, pieces of source code specifying the abstraction (see Figure 4.2). In this sense the fragment serve as a universal representation for other more specific abstraction forms. We will take advantage of this observation to allow programmers to work with intuitive forms of reusable entities, call them abstractions or components, instead of working with fragments explicitly. For users to work with intuitive components they must be

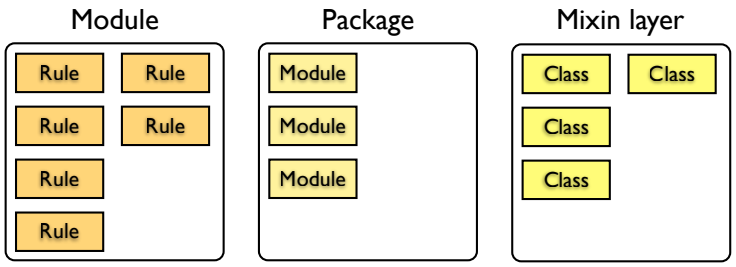


FIGURE 4.2: *Many well-known abstractions are collections of smaller language constructs.*

able to:

1. Define components and describe their interfaces.
2. Make use of defined components by exploiting their interfaces.

We will look at these different issues separately. First, in Section 4.2.1, we look at making fragments more intuitive and in Section 4.2.2 we discuss what domain-appropriate composition operators/statements could be.

4.2.1 Domain-appropriate components

Most programming *units* can be seen as fragments of some sort. By unit we here mean some fundamental block of code that a programmer views as a coherent entity. For example, a Java class is a unit of deployment for a Java programmer. For other programming languages, these units can take different forms. In Chapter 2 we discussed how other kinds of units could be defined by adapting a grammar of a language. For example, we discussed how to allow to define single *rules*, *atoms* or even *variables* as deployable units for our example rule language *RL*. While such fragments in certain circumstances can be desirable to define, for many languages it is often more beneficial to define larger units of deployment. Such larger units could for example be *modules* (collections or rules for a rule-based language), *packages* (collections of modules) or *mixin layers* (collections of collaborating classes for object-oriented languages). See Figure 4.2 for an illustration.

Such reusable entities, components, or abstractions, have an intuitive *raison d'être* for programmers and developers. As can be seen, such larger entities are often collections of other smaller fragment types. In the case of modules for rule-based languages, modules collect sets of related rules. It is furthermore not only important to be able to define such entities, but to do so using some intuitive and informative syntax in order to, in a natural way, encode what is being defined. This can be achieved by extending the considered base language with the needed constructs for defining the reusable entities.

In the following we will consider the rule-based language *RL* from Example 2.1 and the module component type. A *module* is here understood as a collection of related *RL* statements (rules). To provide an intuitive way of defining modules we could add the grammar snipped in Table 4.1 to the base grammar (assuming that $\langle module \rangle$ is a previously undefined nonterminal).

⟨*module*⟩ ::= `MODULE` ⟨*const*⟩ ⟨*stmt*⟩*

TABLE 4.1: *Construct for defining modules for rule-based languages.*

```
1 MODULE sales
2
3 employee(X) :- sales_employee(X).
4
5 sales_employee(steve).
6 sales_employee(marco).
```

LISTING 4.1: *Modules defining the employees in a sales department of a company.*

The two nonterminals that ⟨*module*⟩ is defined in terms of are assumed to already exist in the base grammar (which is the case for the *RL* grammar). The ⟨*const*⟩ nonterminal is here used to provide each module with an identifier (a string). For other base languages and other component types this construct with its syntax will look different, but can be defined in a similar way.

Example 4.1. The rule language module in Listing 4.1 concerns the sales department of a small company. The module defines a set of employees that work in the department and "contributes" this information by stating that they are also employees in a more general sense.

The syntax provided for defining modules, here essentially the keyword `MODULE`, gives intuitive understanding to programmers as to what they are defining.

■

When defining reusable components it is often desirable to associate certain properties with them. An example of such a property is encapsulation. That is, to ensure certain separation between different components when they are composed together. What this means in a particular situation can differ depending on the base language and the desired component type. For our rule language and module concept it would be desirable to ensure that predicates from different modules do not match (or *unify* in logic-programming terminology). This essentially means that rules from different modules should not depend on each other. But for modules to be put together in a useful way when building larger applications it would also be desirable for programmers to be able to explicitly break this encapsulation. That is, to define how modules can communicate and be interfaced.

In our running example this can be achieved by not only providing syntax for defining modules, but also to provide syntax for marking the rules that are part of modules' interfaces. To achieve this we could add the grammar snippet in Table 4.2 to the base language grammar (in addition to the snippet from Table 4.1).

⟨*iface-head*⟩ ::= `@` ⟨*head*⟩

TABLE 4.2: *Construct for defining module interfaces.*

```
1 MODULE sales
2
3 @ employee(X) :- sales_employee(X).
4
5 sales_employee(steve).
6 sales_employee(marco).
```

LISTING 4.2: *The predicate* `employee/1` *is marked as an interface of the module.*

The grammar rule above defines "interface rule heads." To fully integrate this construct into the base grammar we need to make the construct defined by ⟨*iface-head*⟩ an appropriate alternative to "normal" rule heads (represented by nonterminal ⟨*head*⟩). Later in this chapter we will provide a language for doing exactly this. That is, making it easier to appropriately inject such grammar snippets into a base grammar. For the moment we can imagine that nonterminal ⟨*iface-head*⟩ is an additional choice option in the definition of the ⟨*head*⟩ nonterminal (see grammar in Example 2.1).

Example 4.2. The rule language module in Listing 4.2 is the same as in Listing 4.1, but where the rule defining the `employee/1` predicate is marked as an interface statement of the module.

The additional syntax (`@`) is used to declare which rules can be accessed from the outside (that is, from other programs or modules). Thus, in this example all the employees from the sales department can be accessed via the first rule of the module. However, the two facts cannot be accessed directly from the outside, since they are not defined as part of the module interface.

■

In the above we have focused on how simple grammar extensions can provide appropriate syntax for defining intuitive components. We have however not addressed how these additional constructs are handled when executed, that is, their semantics. We shall address these issues in the next section.

4.2.2 Domain-appropriate composition statements

We would like to declare which modules we want to use—that is, import modules—and then be able to reference them in order to make use of the functionality they provide. Accessing modules' functionalities or services must be done using their user-defined interfaces.

Example 4.3. This example will demonstrate how we would like to use modules. Assume that the module in Listing 4.2 can be found at location `file:sales.md`. Then consider the program in Listing 4.3.

The program in Listing 4.3 first imports the previously defined module using the `IMPORT-AS` construct. The construct takes two arguments: the location of the module and a user-given name (here: `sales`). The first rule of the program references this module by using the `IN-MODULE` construct. The `IN-MODULE` construct also takes two arguments: the name of an imported module and an atom within brackets. The `IN-MODULE` construct is thus provided to make use of the module-defined interfaces. The first rule only queries the employees of the sales department, while the second rule only queries the "local" employees (the only local employee here is `john`). Posing the query `bonus(X,Y)` to the program in Listing 4.3 should give the following results:

```
1 IMPORT file:sales.md AS sales
2
3 bonus(X, 200) :- IN sales ( employee(X) ).
4 bonus(X, 100) :- employee(X).
5
6 employee(john).
```

LISTING 4.3: *A program importing and querying a module.*

⟨*import-as*⟩	::=	IMPORT ⟨*file*⟩ AS ⟨*predname*⟩
⟨*in-module*⟩	::=	IN ⟨*predname*⟩ (⟨*atom*⟩)
⟨*file*⟩	::=	LOCATION

TABLE 4.3: *Definition of constructs for importing and using modules.*

$$\{X = steve,\ Y = 200\},\ \{X = marco,\ Y = 200\},\ \{X = john,\ Y = 100\}$$

The query `sales_employee(X)` should give no answers (or an error). The reason is that the queried predicate is not available, but is encapsulated in the referenced module. The query `employee(X)` would give the single answer:

$$\{X = john\}$$

since only the local employees are directly accessible.

■

Using the modularization constructs as demonstrated in Examples 4.2 and 4.3 is not only helpful for programmers, but doing so is also intuitive. The program in Listing 4.3 can be authored by augmenting the base grammar with appropriate constructs. For example, by introducing the grammar snippet in Table 4.3. The ⟨*predname*⟩ and ⟨*atom*⟩ nonterminals are assumed to be defined in the base grammar, which they are for *RL*. The ⟨*predname*⟩ nonterminal is here chosen to represent the module name, but a different nonterminal could also be chosen as long as it generates the desired set of strings. Again, for proper integration the constructs defined in Table 4.3 need to be appropriately injected into the base grammar. For example, the `IMPORT-AS` construct should be a valid statement and the `IN-MODULE` construct must be a valid alternative to ⟨*atom*⟩.[2]

The realization of the semantics of the examples shown above can be achieved by composition. That is, the programs and modules can be composed together such that the intended semantics is maintained, for example, such that the intended module encapsulation is enforced.

Charles W. Krueger describes abstractions by making a clear separation between the abstraction *specification* and its corresponding *realization* [60]. Using this vocabulary we understand that the constructs `MODULE`, `IMPORT-AS` and `IN-MODULE` provide

[2] As a remark, such injections might make the language generated by the extended grammar unintentionally large. For example, due to the structure of the base grammar we might be allowed to use the `IN-MODULE` construct in the head of rules, something which is not intended. For a discussion, see Section 4.6.

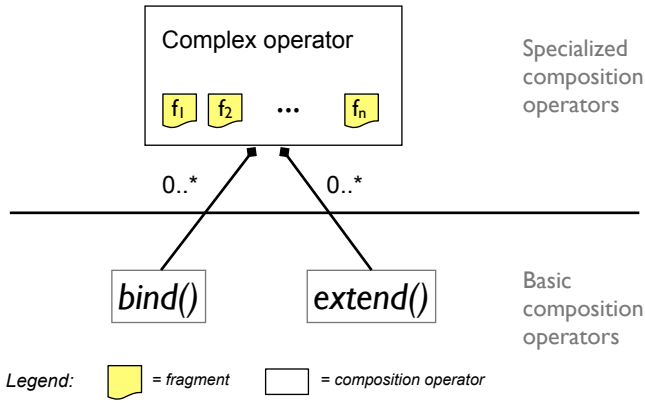


FIGURE 4.3: *A complex composition operator is defined in terms of the more basic operators* bind() *and* extend() *and can also contain internal fragments.*

means to specify the module abstraction, while its composition would constitute the corresponding realization. The exact realization technique is a design issue and can be very different for varying component types and base languages. There can also exist different realization techniques for the same language and component type. In the context of our composition framework, it is important to remember that the abstraction realization must be encoded in the base language. We recall that our framework dictates that the composition results must belong to the languages generated by the base grammars. Hence, an abstraction realization must be programmable using the base language itself. For example, the discussed module encapsulation must be encoded in instances of the *RL* language. This could for example be achieved by renaming predicate names, or by more fanciful inventions.

In the following section we will discuss how to use ISC and our defined framework to make the connection between introduced abstraction *specification* constructs and implementation procedures for the abstraction *realization*.

4.3 Domain-appropriate composition operators

The realization of an abstraction—appropriate transformations to handle the component type and ensure certain properties such as encapsulation—can often be a non-trivial task. However, the overall transformation task can often be broken up into a number of smaller tasks. These smaller transformation tasks, or composition steps, can be executed by ISC's basic composition operators *bind()* and *extend()*. Thus, the full realization of an abstraction can then be described by a set of collaborative calls to ISC's basic operators. We call such a set a *complex composition operator*, or just *complex operator* (see Figure 4.3 for an illustration).

A complex operator is atomic and always assumed to be executed in its entirety, or not at all. It should also be noted that a complex operator does not only contain calls to ISC's basic operators, but can also contain *internal fragments* that are needed for the realization of the abstraction the operator is implementing (see fragments $f_1, \ldots, f_n$ in Figure 4.3). An example of such an internal fragment could be a fragment containing

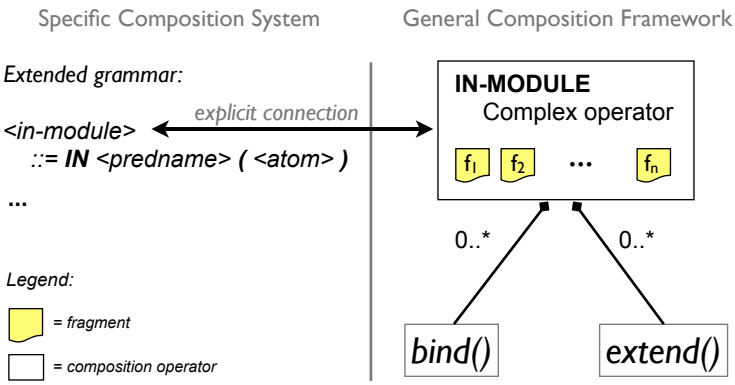


FIGURE 4.4: *A complex composition operator is connected to an active syntax construct in an extended grammar.*

an identifier (a name) which is used in some renaming scheme during composition (for example, for renaming predicate names for our above-discussed modules).

In Section 4.2 we made the distinction between domain-appropriate components and domain-appropriate composition statements. It can be said that this overall domain-appropriateness was achieved by adorning the base language with additional syntax for the purpose of component-based development. We can separate these syntactical adornments into the following two categories:

1. *Passive syntax.* This is syntax that is used by the programmer to define components or describe how they should be used. For our rule-based modules the `MODULE` and `@` (module interface) constructs are examples of passive syntax.

2. *Active syntax.* This is syntax that is used to deploy components and that takes an active part in the composition of those components. For our rule-based modules the `IMPORT-AS` and `IN-MODULE` constructs are examples of active syntax.

The notion of active syntax is closely related to complex composition operators. The relation is that an active syntax construct (e.g. `IMPORT-AS`) delegates its work to a complex composition operator. Or seen the other way around, a complex composition operator implements an active syntax construct. The passive syntax constructs are not composition operators, but are used to guide the composition and hence play an equally important role.

The relation between an active syntax construct and a complex composition operator has to be made explicit by a developer. This is done on the grammar level. For example, since the `IN-MODULE` construct defined above is an active syntax construct, it should be connected to some complex composition operator implementing its functionality. This is illustrated in Figure 4.4.

A complex composition operator can use our introduced framework to implement the intended semantics of its corresponding active syntax construct. This allows the complex operator to make use of both explicit and implicit fragment interfaces in its implementation, and to use ISC's generic composition algebra. This essentially means that a complex composition operator is a special kind of composition program. The only difference being that this kind of composition program takes some external input,

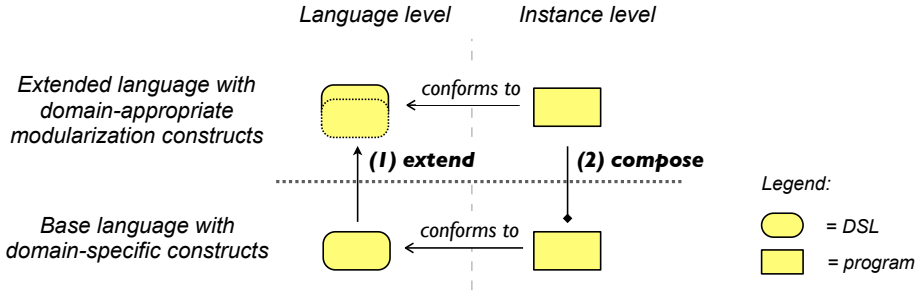


FIGURE 4.5: *A language (in particular a DSL) can be (1) extended to provide for certain modularization constructs. By defining special composition programs in the style of ISC we can then (2) compose programs of the extended language into semantically equivalent programs of the non-extended language.*

in form of the fragments it is supposed to be working on. This external input directly relates to the definition of the active syntax construct that the complex operator is implementing. For example, the `IN-MODULE` construct from Table 4.3 would need to know the name of the referred module, as well as the atom querying the module, represented by nonterminals ⟨*predname*⟩ and ⟨*atom*⟩, respectively. Since we are using Java as our host composition language, these special composition programs can be realized as Java methods. Considering our module extension to *RL* and an appropriately generated core composition language, the signature of a complex composition operator method implementing the `IN-MODULE` construct would be (possibly using a different name):

```
public IAtom inModuleOperator(IPredname name, IAtom atom)
```

The parameter types directly correspond to the definition of the `IN-MODULE` construct. The fragment returned from a complex composition operator replaces the active syntax that invoked it. So, the return type must belong to the set of (grammatical) types of the location of the active syntax. In the above the return type is specified to be `IAtom`. This will be type correct since the `IN-MODULE` construct was defined as an alternative to *atom*s (represented by the core composition language type `IAtom`). Hence, an `IAtom` can replace an `IN-MODULE`. In Section 4.4 we will see how these complex operators can be defined in practice, and how they explicitly can be related to nonterminals in an extended grammar.

Summary

Let us recall our objectives and summarize the suggested approach. We are driven by two issues:

1. Address the important problem of finding a universal way of enabling component-based development for DSLs.
2. Realize the vision for ISC of providing "domain-specific, tailored and adequate" composition operators for users of composition systems.

We argued that the 'fragment' is not the ideal abstraction. Instead we are aiming for more domain-appropriate abstractions. Such abstractions can arguably be achieved and

allowed to be specified by appropriate language extensions (cf. Section 4.2). Hence, we propose to employ small language extensions to enable "domain-appropriate" and "tailored" composition opportunities, essentially to be able to define suitable component types ("abstraction specifications" in the words of Krueger). Then, by exploiting the composition technology underlying ISC and our framework, we allow for the definition of "adequate" composition operators ("abstraction realizations" in the words of Krueger) that composes programs of the extended language into semantically equivalent programs of the non-extended base language. This is illustrated in Figure 4.5. By doing this we can address both issues enumerated above.

4.4 Developing E-ISC–based composition systems

In the previous chapters and sections we have discussed—quite independently—several different issues: how to adapt a language to the basic ideas of ISC, how to write composition programs addressing both explicit and implicit interfaces, how to extend the language's grammar with domain-appropriate and component-oriented constructs, and finally how to define complex composition operators. In this section we intend to combine our solutions and explain how they can be brought together for developing a domain-appropriate composition system.

First, in Section 4.4.1, we introduce an extension of the component model specification language *CmSL*. In Section 4.4.2 we then describe the overall composition system development process. Finally, in Section 4.5, we give a concrete example where we develop a composition system for our example rule language *RL*.

4.4.1 Extended component model specification language ($CmSL^+$)

This section describes an extension of the *CmSL* language from Section 3.3.1, called, $CmSL^+$. The extended language, $CmSL^+$, needs to be able to do three main things that was not possible with *CmSL*:

1. Define grammar snippets that correspond to the abstract syntax of required language extensions. The constructs defined in a language extension are intended to be used to specify the desired abstractions.

2. Define how the newly defined constructs relate to the base grammar.

3. Separate between *active* and *passive* syntax. Active syntax constructs are marked such that the resulting composition system knows when complex composition operators are to be executed.

These requirements are addressed by the $CmSL^+$ constructs *definition*, *injection* and *annotation*, respectively. Thus, beyond the constructs in *CmSL*, the following are provided by $CmSL^+$:

– **Construct definition.** It should be possible to define new domain-appropriate constructs, such that components and compositions can be specified in an intuitive way. Traditional EBNF-like grammar snippets can be defined using the *definition* construct:

⟨*definition*⟩ ::= ⟨*nonterminal*⟩ **::=** ⟨*nonterminal*⟩ (**,** ⟨*nonterminal*⟩)* **.**

where the first nonterminal is not previously defined in the base grammar, but where the nonterminals to the right of `::=` (separated by `,`) might belong to the base grammar (if a nonterminal is not defined in the base grammar, it must be defined in the same specification). Notice that in the above we only consider abstract syntax. Concrete syntax is also possible to specify, but is here left out for simplicity reasons.[3] As in EBNF, the nonterminals to the right of `::=` (separated by `,`) can also be annotated with cardinality restrictions (i.e. `*`, `+`, or `?`). We can also define choices:

⟨*definition*⟩ ::= ⟨*nonterminal*⟩ `::=` ⟨*nonterminal*⟩ (`|` ⟨*nonterminal*⟩)* .

where all the nonterminals to the right of `::=` (separated by `|`) have been defined by the *definition* construct, and are not annotated with cardinality restrictions.

– **Construct injection.** When defining new constructs, they need to be "injected" into the base language such that they can be used. By "injection" we here mean intrusive re-definition of base language constructs to make way for newly defined constructs. For this purpose, the *injection* construct is used:

⟨*injection*⟩ ::= ⟨*nonterminal*⟩ <> ⟨*nonterminal*⟩ .

where the first nonterminal is defined using the *definition* construct introduced above, and the second nonterminal is defined in the base grammar. Using the *injection* construct makes the left-hand side nonterminal a valid alternative for the right-hand side (already defined) nonterminal. This injection is done in a similar way to how slots were introduced into grammars (cf. Section 2.2.1).

– **Construct annotation.** Nonterminals defining constructs that correspond to active syntax must be annotated using the *annotation* construct:

⟨*annotation*⟩ ::= ⟨*nonterminal*⟩ `->` `@Composer` .

The nonterminal is assumed to have been defined using the *definition* construct described above. Annotating a nonterminal in this way lets the composition system know that a complex composition operator is to be executed when the annotated construct is used in composition programs or components.

– **Fragment types.** The `fragtypes` construct from the *CmSL* language is extended to not only allow nonterminals from the base grammar, but also nonterminals that are defined in the same specification (using the *definition* construct introduced above).

The following are some additional notes about the above constructs, mainly related to the REUSEWARE and REUSEWARE/AIR realizations, but presented here to be able to provide concrete examples:

– *Base language references.* In the above, nonterminals that refer to the base grammar are preceded by a reference to the base grammar. For example:

```
base.NonTerminal
```

[3] This is made possible by REUSEWARE (cf. Section 3.4).

where `base` has been associated with the base grammar using the `extends` construct from *CmSL*.

– *Abstract syntax role names.* As mentioned, the *definition* construct is only used to specify abstract syntax. To make it easier to annotate abstract syntax specifications with concrete syntax, each nonterminal in a *definition* construct is preceded by a *role name*. This looks as follows:

```
rolename:NonTerminal  or  rolename:base.NonTerminal
```

depending on whether a newly defined nonterminal, or a base grammar nonterminal, is referenced, respectively. This is done to be compatible with the REUSEWARE tooling and framework on which these language extensions rely (cf. Section 3.4). This technique of separating abstract and concrete syntax using role names is detailed in [65, Chapter 3].

– *Automatically resolved locations.* In component-based development, it is often the case that the basic units of discourse reside in separate files on the file system, or as URLs on the Web. Consider, for example, our *RL* modules. Intuitively, it is desirable to store each module definition in a separate file (cf. Listing 4.2). While defined separately, module definitions will be integrated into larger programs during deployment. We recall that modules should be encapsulated, and it is the task of the module import construct (`IMPORT-AS`) to achieve this. When using the `IMPORT-AS` construct, we want to refer to a location, rather than a module definition:

```
IMPORT file:module.dl AS mod
```

But the `IMPORT-AS` construct is active syntax and its semantics will be implemented by a complex composition operator. Since this complex operator will transform a module definition, it is appropriate that it receives a module definition as input, rather than the location of the module. As mentioned, in our realization, complex composition operators will be implemented as Java methods. An appropriate signature for a Java method implementing the active syntax construct `IMPORT-AS` would be:

```
IStatement importAs(IModule mod, IPredname name)
```

where `mod` is the imported module definition, and `name` is the shorthand name for the module. Clearly, there is a gap between how a construct is to be used, and how it is to be implemented. To close this gap, and make the notion of a "location" transparent—something automatically resolved by the system—we allow nonterminals in *definition*s to be annotated to say that they represent constructs that actually will reside in separate files. This is exemplified below for the definition of the `IMPORT-AS` construct (defining nonterminal `ImportAs`):

```
ImportAs = location:Module [ @Location ], name:rl.Predname .
```

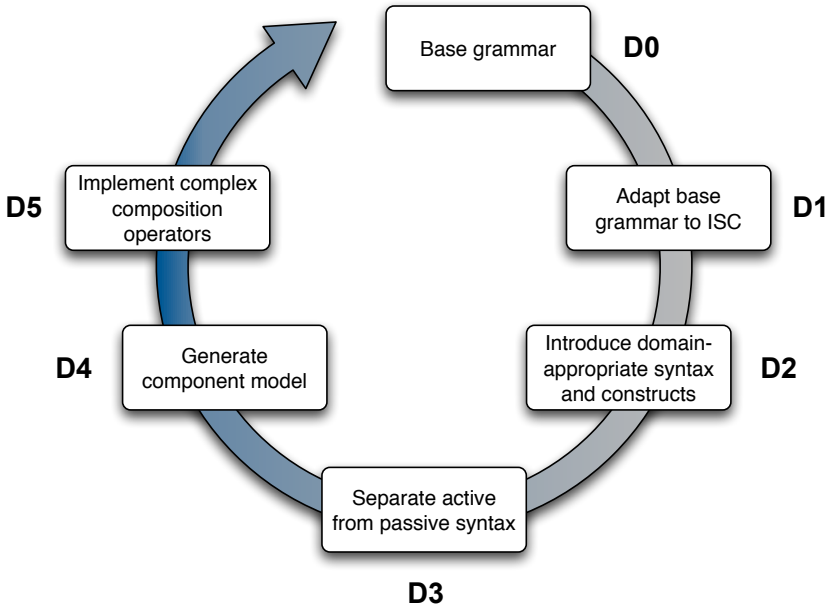


FIGURE 4.6: *The process steps for developing a composition system for a particular language.*

where `Module` is the nonterminal defining module definitions. This will automatically create a language construct which, syntactically, expect a location, rather than a module definition as one of its arguments. However, when calling the implementing composition operator method, this location is automatically resolved into a fragment of the appropriate type (`Module`, or `IModule` in the generated core composition language).

Extended Component Models Extending a language for the purpose of component-based development in this way can be seen as defining extended component models wrt. the component models as defined in Chapters 2–3. These extended component models do not only cover the primitive fragment interfaces from GBM and ISC, but captures higher-level types of interfaces. This especially holds true for passive syntax. That is, extended component model specifications may introduce (passive) syntax for defining component interfaces that are intuitive to understand for end-users (cf. the `@` interface construct for *RL*). Active syntax constructs certainly complements these interfaces by providing constructs for using them. We shall again see examples of this in Sections 4.4.2–4.5.

4.4.2 Development process

Below we describe the complete process of developing a domain-appropriate composition system, or an embedded invasive software composition (E-ISC) system. We will demonstrate the process using our example rule language *RL* and augment it with the already discussed module component type.

The illustration in Figure 4.6 shows the main steps required to develop a composition system that addresses a particular language. The main steps are:

D0 *Base grammar.* Since the development process is *grammar-driven*, it always starts with a base grammar, specifying the language of interest.

D1 *Adapt the grammar to ISC.* This step essentially involves specifying how fragments are allowed to look, by introducing slot constructs, and how they are allowed to be transformed by providing grammar annotations for certain nonterminals. This is done using the C_mSL^+ language for component model specification (or the more basic C_mSL language).

D2 *Introduce domain-appropriate syntax and constructs.* This step involves introducing both passive and active syntax for the benefit of the programmer. These constructs should correspond to the language extension being considered. This is done using the C_mSL^+ language. Furthermore, it should be specified how the newly defined constructs relate to the base language by using the *injection* construct from C_mSL^+.

D3 *Separate active from passive syntax.* It is important to distinguish the two, since they play different roles during composition. Active syntax must be annotated using the *annotation* construct from C_mSL^+.

D4 *Generate component model.* When the full component model has been specified in steps $D1 - D3$, the component model can be generated. This will trigger the core composition language to be generated.

D5 *Implement complex composition operators.* The extended constructs corresponding to active syntax must be given compositional semantics, which is done by implementing their corresponding complex composition operators. This can be done since the core composition language was generated in the previous step.

To make things clearer we shall go through the above steps using the rule language *RL* as the base language and extend it with the concept of modules.

4.5 Example: E-ISC–based composition system

Here we go through the different development steps for our example rule language *RL*.

D0 *Base grammar.* Our base grammar is *RL*'s grammar. However, we here specify a variant which more closely resembles the grammar that is running in our demonstrator. We here only provide the abstract syntax, but the concrete syntax can be assumed to be the one used in all examples throughout Chapters 2–4.

$\langle RL \rangle$ ::= $\langle Unit \rangle$
$\langle Unit \rangle$ ::= $\langle Program \rangle$
$\langle Program \rangle$ ::= $\langle Statement \rangle$+
$\langle Statement \rangle$ ::= $\langle Rule \rangle$ | $\langle Fact \rangle$ | $\langle Comment \rangle$
$\langle Rule \rangle$ ::= $\langle Head \rangle$ $\langle Body \rangle$
$\langle Fact \rangle$ ::= $\langle Head \rangle$

```
extends file:rl.gr @ rl as file:rrl.gr .

% i) passive syntax
Module            =  moduleName:rl.Predname, moduleStmt:rl.Statement* .
Module            <> rl.Unit .

OutInterface      =  interface:rl.Head .
OutInterface      <> rl.Head .

% ii) active syntax
ImportAs          =  moduleLocation:Module [ @Location ],
                     moduleName:rl.Predname .
ImportAs          <> rl.Statement .
ImportAs          -> @Composer .

InModule          =  moduleName:rl.Predname, interface:rl.Atom .
InModule          <> rl.Atom .
InModule          -> @Composer .

% iii) fragment types
fragtypes { rl.Program, rl.Statement, rl.Rule, rl.Head, rl.Atom, rl.Variable,
            rl.Predname, Module, OutInterface }
```

LISTING 4.4: C_mSL^+ *specification for extending the rule language* RL *with constructs supporting the notion of a 'module.'*

$$
\begin{aligned}
\langle Head \rangle &::= \langle Atom \rangle \\
\langle Body \rangle &::= \langle Atom \rangle + \\
\langle Atom \rangle &::= \langle Predname \rangle \ \langle Term \rangle + \\
\langle Term \rangle &::= \langle Variable \rangle \mid \langle Constant \rangle \mid \langle Num \rangle \\
\langle Predname \rangle &::= \text{STRING} \\
\langle Variable \rangle &::= \text{CAP\_STRING} \\
\langle Constant \rangle &::= \text{STRING} \\
\langle Num \rangle &::= \text{NUM\_STRING} \\
\langle Comment \rangle &::= \text{STRING}
\end{aligned}
$$

D1-3 *Adapt base grammar to ISC, define domain-appropriate constructs and separate out active syntax.* The C_mSL^+ specification in Listing 4.4 extends the base grammar for the purpose of working with modules.

The component model specification in Listing 4.4 achieves three things: i) It defines passive syntax for defining rule modules and their interfaces, ii) It defines active syntax for importing modules (`IMPORT-AS`) and for querying modules (`IN-MODULE`), and iii) It allows constructs corresponding to a set of nonterminals to be defined as fragments, as well as to be accessed implicitly during composition. The concrete syntax for the newly defined constructs is not given here, but can be assumed to be the one used in Section 4.2.

The language extension specified above will allow to author programs such as can be found in Listings 4.2 and 4.3.

D4 *Generating component model.* The next step is to generate the component model. This essentially involves generating the core composition language, as described

```
1 IMPORT file:sales.md AS sales
2
3 bonus(X, 200) :-
4       IN sales ( employee(X) ).
5 bonus(X, 100) :- employee(X).
6
7 employee(john).
```

LISTING 4.5: *Rule program importing a module.*

```
1 MODULE sales
2
3 @ employee(X) :-
4       sales_employee(X).
5
6 sales_employee(steve).
7 sales_employee(marco).
```

LISTING 4.6: *Rule module at* `file:sales.md`.

in Section 3.3.2. This will generate the appropriate Java types for implementing the complex composition operators corresponding to the active syntax constructs as defined in Listing 4.4.

D5 *Defining composition operators for active syntax constructs.* The next step in the development process is to associate composition operators with the active syntax constructs introduced in the previous steps. We notice that there are two such constructs, defined by the nonterminals `ImportAs` and `InModule`. Hence, we must define composition operators for these constructs. They have the following signatures (method names can vary):

```
IStatement   importAs(IModule mod, IPredname name)
IAtom        inModule(IPredname mod, IAtom atom)
```

Notice that the signature of each operator can be derived from the C_mSL^+ specification in Listing 4.4. The *definition* constructs define the parameters, while the *injection* constructs define the return types. In general, the return type could be different from what is specified using the *injection* construct. For example, we could inject an active syntax construct as an alternative for *RL*'s `Term`, but return a `Var` from the implementation method corresponding to the active syntax construct. What is important, however, is the following: Since the result of a composition operator replaces the active syntax that invoked it, this replacement must be valid wrt. the extended language. Hence, the safety conditions from Chapter 2 still apply.

The definition of the operators can be found in Listing 4.8 in Appendix 4.A.

Execution of composition In the following we look at how the above specified composition system is used and how the involved fragments are transformed. That is, we try to give an intuitive understanding of how the *RL* module system is realized, and how the defined composition operators work. Assume we have the programs shown in Listings 4.5 and 4.6.

The program in Listing 4.5 is a program that imports the module in Listing 4.6. Intuitively, the programs in Listings 4.5 and 4.6 specify an abstraction. The goal of the composition is to define the abstraction realization. This abstraction realization should ensure properties associated with the abstraction. One such property is encapsulation. We choose to realize the encapsulation via renaming of predicate names:

1. **Module encapsulation.** This transformation scheme is used for atoms in modules that are not part of the module interface, that is, local atoms. The following string is added as a suffix to the predicate names of such atoms:

```
<Module>
 <stmt>
  <rule>
   <InterfaceOut>
    <head>
     <atom>
      <const>employee</const>
      <term><var>X</var></term>
     </atom>
    </head>
   </InterfaceOut>
   <body>
     <atom>
     <const>
      sales_employee
     </const>
     <term><var>X</var></term>
    </atom>
   </body>
  </rule>
 </stmt>
 <!-- snip -->
</Module>
```

```
<Module>
 <stmt>
  <rule>
   <head>
    <atom>
     <const>
      employee_sales_out
     </const>
     <term><var>X</var></term>
    </atom>
   </head>
   <body>
    <atom>
     <const>
     sales_employee_sales_priv
     </const>
     <term><var>X</var></term>
    </atom>
   </body>
  </rule>
 </stmt>
 <!-- snip -->
</Module>
```

FIGURE 4.7: *Part of the AST of the fragment in Listing 4.5. The left-hand side before composition, the right-hand side after composition (to ensure encapsulation).*

```
_[module name]_priv
```

2. **Module interfaces.** This transformation scheme is used for atoms that are part of module interfaces. This holds both for the head atoms in statements proceeded by the `@` module interface construct, and for body atoms used with the `IN-MODULE` construct. The following string is added as suffix to the predicate names of such atoms:

```
_[module name]_out
```

The above renaming scheme is quite simple and could be replaced by a more involved method, but this suffices here. The renaming scheme is encoded in the composition operators that are used to define the composition. The importing program (Listing 4.5) can be seen as a composition program that is connected to composition operators via its use of active syntax constructs, which triggers the composition. As there are two active syntax constructs in the composition program, two composition operators will be executed. The first operator to be executed is the `IMPORT-AS` operator, which corresponds to the first operator specified in Listing 4.8. The operator receives two arguments, the definition of the module that is to be imported, and the shorthand name for the module. The module is then transformed using its implicit interface. Part of the transformation of the module's AST is shown in Figure 4.7. The left-hand side of Figure 4.7 shows the original AST, while the right-hand side shows the transformed AST.

The transformed set of statements of the module are returned from the composition operator and replaces the `IMPORT-AS` construct of Listing 4.5. In the same manner, the call to the `IN-MODULE` operator in Listing 4.5 transforms the atom being passed

```
1 % -- module --
2 employee_sales_out(X) :- sales_employee_sales_priv(X).
3
4 sales_employee_sales_priv(steve).
5 sales_employee_sales_priv(marco).
6
7 % -- main program --
8 bonus(X, 200) :- employee_sales_out(X).
9 bonus(X, 100) :- employee(X).
10
11 employee(john).
```

LISTING 4.7: *The result of composing the programs in Listings 4.5 and 4.6.*

to it according to the second operator in Listing 4.8 (p. 126), before replacing itself with the result from the operator. The composed result is shown in Listing 4.7 (with transformed statements in italics, and comments added for clarity).

The rule program in Listing 4.7—valid wrt. the original *RL* grammar—is thus the realization of the abstraction used in Listings 4.5 and 4.6. This realization is not intended to be seen or worked on by programmers directly, but represents the program that will be interpreted by the rule engine developed for the base language.

Roles in composition systems. As we mentioned in the introduction of this chapter, we aimed to separate between a *developer* who has to understand the details of ISC, and a *user* who can take advantage of the benefits of ISC, without having to understand all the intricate details of fragments, their interfaces and primitive composition operators. This is the general goal of embedded ISC (E-ISC) systems. By specifying composition systems in C_mSL^+ and implementing composition operators for active syntax constructs using a generated core composition language, exactly this can be achieved. The developer of the embedded composition system has to use C_mSL^+ and write the composition operators (cf. Listing 4.8, p. 126). The user on the other hand can write program with intutive and appropriate syntax, programs like the ones in Listings 4.5 and 4.6.

4.6 Summary and Discussion

Let us recap what was achieved and discussed in this chapter, as well as point to some open issues.

1. *A remedy for the primitiveness of ISC.* We recognized the primitiveness of ISC's composition operators and concluded that working on that level of detail is not optimal, or even desirable, for programmers. To address this situation, we also realized the need to separate between different user roles wrt. composition systems. We distinguish between composition system developers, and composition system users. This has the effect that users can employ ISC that is "embedded" in their normal languages. The embedded ISC provide intuitive abstraction constructs that the users only have to know how to use, not how they are realized using ISC. The developer on the other hand has to know how to realize the same abstraction constructs, that is, how to implement their semantics.

2. *Domain appropriateness.* We identified the need to have a generic approach to working with fragments (in order not to exclude languages), but also to have the ability to connect this generic approach with language-tailored and more specific constructs suitable for certain languages. This led to the notion of extending a base language with two kinds of syntax:

   (a) Passive syntax – this is syntax that is intended to make it more natural to define "components" and their interfaces, whatever this might mean for a particular language.

   (b) Active syntax – this is syntax that correspond to composition operators, statements or expressions that appropriately transform components, that is, implement composition operator semantics (or component realization).

3. *Complex composition operators.* To bridge any domain-appropriate constructs that are introduced into a base language with the general composition approach provided by universal ISC, we introduced the concept of complex composition operators. Complex operators connect active syntax constructs with particular kinds of composition programs. Since active syntax constructs trigger the execution of complex operators, programs written in an extended language can be seen as composition programs by the framework.

4. *Embedded invasive software composition.* The overall achievement of this chapter was to introduce embedded invasive software composition (E-ISC). Using all the notions mentioned above, E-ISC makes it possible to abstract from the primitiveness of ISC and its universal notion of fragments.

5. *Extended component model specification language* — $\mathrm{C_mSL^+}$. We provided an extended component models specification language – $CmSL^+$. This language can be used to specify components models for embedded ISC systems.

6. *Development process.* We described the overall development process for creating embedded ISC systems.

As a concluding remark, it can be said that most DSLs provide appropriate constructs for "programming in the small" [27]. That is, they provide constructs for developing the core parts of their programs. However, as discussed, many DSLs lack constructs for "programming in the large," or at least language-appropriate constructs for doing so (if the DSLs is embedded, it has to use whatever is provided by the host language, which might not always be ideal). Nonetheless, such "programming in the large" constructs are often needed for organizing the smaller parts of a larger program in the appropriate way. Furthermore, it enables reuse of already specified program parts. With our embedded ISC approach it is possible to provide these "programming in the large" opportunities after the core language is already specified and developed. We are here mainly thinking of the non-embedded DSLs that do not even have the opportunity of reusing constructs from a host language. In these embedded ISC systems, ISC and its composition technique is used as a core component. In addition, with this approach we have in a sense acknowledged the vision of the need for more "domain-specific, tailored and adequate" composition operators [5, p. 278], and we exemplified how this insufficiency can be remedied.

Discussion

Below we discuss certain points of interest:

1. *Unintended language extensions.* Injecting new constructs into a base language can sometimes make the extension larger than was intended. As an example, consider our rule language example and its module extension. Suppose part of the *RL* grammar looks as follows:

   ⟨*rule*⟩ ::= ⟨*head*⟩ `:-` ⟨*body*⟩ `.`
   ⟨*head*⟩ ::= ⟨*atom*⟩
   ⟨*body*⟩ ::= ⟨*atom*⟩ (`,` ⟨*atom*⟩)*
   ⟨*atom*⟩ ::= ⟨*predname*⟩ `(` ⟨*term*⟩ (`,` ⟨*term*⟩)* `)`

   Now, suppose we want the `IN-MODULE` construct from the module extension to be an alternative for *atom*s in rule bodies. If we inject `IN-MODULE` as an alternative for *atom*, we would get something like this:

   ⟨*atom*⟩ ::= ⟨*atom'*⟩ | ⟨*in-module*⟩
   ⟨*atom'*⟩ ::= ⟨*predname*⟩ `(` ⟨*term*⟩ (`,` ⟨*term*⟩)* `)`

   However, this would also allow `IN-MODULE` constructs to appear in rule heads, which we do not want. Hence the extension is too large. A solution to avoiding this would be to modify the base grammar specification. We could modify the definition of the ⟨*body*⟩ nonterminal and introduce an explicit construct representing things in rule bodies, something like:

   ⟨*body*⟩ ::= ⟨*body-part*⟩ (`,` ⟨*body-part*⟩)*
   ⟨*body-part*⟩ ::= ⟨*atom*⟩

   Then we could say that ⟨*in-module*⟩ should be injected as an alternative for ⟨*body-part*⟩s. We would get something like:

   ⟨*head*⟩ ::= ⟨*atom*⟩
   ⟨*body*⟩ ::= ⟨*body-part*⟩ (`,` ⟨*body-part*⟩)*
   ⟨*body-part*⟩ ::= ⟨*atom*⟩ | ⟨*in-module*⟩

   In this case, the `IN-MODULE` construct would only be allowed to appear in rule bodies, which is what we wanted. In rule heads, only *atom*s are allowed.

   As can be seen, it is important to be aware of how the specification of the base grammar affects any transformations done to it by component model specifications. This again shows how the approach really is grammar-driven: The specification of the base language grammar influences the construction and use of the resulting composition system.

2. *Development process.* In Section 4.4.2 we described all the necessary steps in the process of developing an E-ISC–based composition system. However, that description was based on a 'perfect' development process, and not on the iterative process that is more likely to occur in real life. Based on our experiences in developing composition systems using our framework, the process illustrated in Figure 4.8 is closer to what a developer would encounter. This development process consists of two parts:

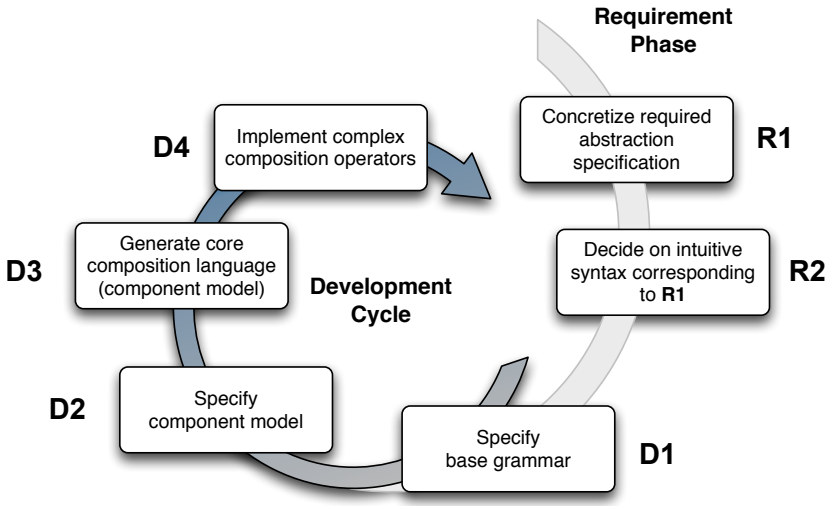


FIGURE 4.8: *A refined development process consisting of a* requirement phase *and a* development cycle. *The requirement phase determines the desired abstraction specification, while the development cycle determines the realization of the same.*

(a) *Requirement phase (R1–2).* The requirement phase consists of deciding what the desired language abstraction is. That is, what the new abstraction is that should enable programmers to be more productive, better understand large specifications and reuse more of their code. In particular, how programmers should concretely *specify* that abstraction in the most intuitive way.

(b) *Development cycle (D1–4).* The development cycle then determines how the abstraction specification should be *realized*. This first involves specifying the base grammar and the component model. Then the core composition language is generated, whereupon complex composition operators can be implemented. These steps form an iterative cycle. For example, a certain fragment type might not be definable, but is found out to be required for the implementation of a composition operator. Hence, the component model specification must be changed, and the core composition language regenerated. In some cases, the base grammar might have to be restructured. This cycle goes on until the abstraction realization corresponds to the abstract specification determined in the requirement phase.

3. *Trade-off between syntax and composition operators.* There can be a trade-off between intuitive abstraction specification syntax and how easy it is to realize the abstraction. That is, having a very intuitive extended language syntax might make the implementation of composition operators more complicated. And vice versa, a simpler implementation of the composition operators might require the extended syntax to be compromised with. Developers of E-ISC–based composition systems should be aware of this trade-off.

4. *Composing extension to base language.* The framework is based on the fundamental idea that a program $P+$ written in an extended language $L+$ is trans-

formed into an equivalent program P of the base language L (where $L+$ is an extension of L). But more importantly, that the properties associated with any components used in $P+$ are properly handled in P. One can question if this assumption and approach is a reasonable one. We argue that it indeed is a reasonable approach, and base this on two observations:

(a) Charles W. Krueger explains in [60] that abstractions, among other things, can be seen from the perspective of a *specification* and a *realization*. The specification is the view of the programmer, that is, its use, while the realization is the implementation of the abstraction. In our case, components written in the extended language $L+$ are abstraction specifications, while the same components in L after composition are their respective realizations.

(b) Clemens Szyperski explains in [86, p. 10] that: "[...] from a purely formal point of view, there is nothing that could be done with components that could not be done without them."

These two observations convince us that the extensions we are interested in—extensions for abstraction *specifications* as explained in 4a—do not need added expressiveness for their *realization*, due to the explanation in 4b.

4.A Appendices

The complex composition operators used in the modular extension of *RL* can be found in Section 4.8. The constructs (active syntax) that prompt the execution of the operators are demonstrated in, for example, Listing 4.5. The `IMPORT-AS` construct corresponds to the method named `importInterpreter`, and the `IN-MODULE` construct corresponds to the method named `inModuleInterpreter`. The code makes use of the core composition language (Java API) generated from the component models specification in Listing 4.4 (these Java classes have the namespace `org.reuseware.air.language.rl`).

Composition operators for *RL* modules

```
1  package org.reuseware.air.language.rl.ops;
2
3  import java.util.Hashtable;
4
5  import org.reuseware.air.algebra.fragment.FragmentSystem;
6  import org.reuseware.air.coconut.IComplexOperator;
7  import org.reuseware.air.coconut.ReusewairComposer;
8  import org.reuseware.air.language.rl.IAtom;
9  import org.reuseware.air.language.rl.IHead;
10 import org.reuseware.air.language.rl.IModule;
11 import org.reuseware.air.language.rl.IOutInterface;
12 import org.reuseware.air.language.rl.IPredname;
13 import org.reuseware.air.language.rl.IStatement;
14 import org.reuseware.air.language.rl.algebra.RlVisitor;
15 import org.reuseware.air.language.rl.impl.IHeadImpl;
16 import org.reuseware.air.language.rl.impl.IPrednameImpl;
17 import org.reuseware.air.language.rl.impl.IStatementImpl;
18
19 import de.tudresden.reuseware.language.rl.RlPackage;
20 import de.tudresden.reuseware.language.rrl.RrlPackage;
```

```
21
22 public class Composers implements IComplexOperator {
23
24     // internal fragments
25     private static IPredname interfacePrivate;
26     private static IPredname interfaceOut;
27     private static String sep = "_";
28
29     // communication information between operators
30     static Hashtable<String,String> names = new Hashtable<String,String>();
31
32     /**
33      * Required by IComplexOperator
34      *
35      */
36     public void initialize() {
37
38         FragmentSystem.getInstance().setGrammar("rrl");
39         // clear names
40         names.clear();
41     }
42
43     public Composers() {
44
45         FragmentSystem.getInstance().setGrammar("rrl");
46
47         interfacePrivate = IPrednameImpl.load("priv");
48         interfaceOut = IPrednameImpl.load("out");
49     }
50
51     /**
52      * IMPORT STATEMENT COMPOSER
53      *
54      */
55     @ReusewairComposer("ImportAs")
56     public static IStatement importStmt(IModule module, final IPredname name) {
57
58         // 1) Extract module statements
59
60         IStatement stmt = new IStatementImpl();
61         module.accept(new RlVisitor(stmt) {
62
63             public boolean visit(IStatement node) {
64                 getParamFragment().extend(node);
65                 return true;
66             }
67
68             /**
69              * For composer communication
70              *
71              */
72             public boolean visit(IPredname node) {
73
74                 if (node.inContextOf(RrlPackage.Literals.MODULE) &&
75                         !node.inContextOf(RlPackage.Literals.STATEMENT)) {
76                     /**
77                      * Save the connection between the prefix name and
78                      * the name as defined by the module
79                      */
80                     if (!names.containsKey(name.toString()))
81                         names.put(name.toString(), node.toString());
82                 }
```

```
83              return true;
84          }
85      });
86
87      // 2) Transform module statements
88
89      // encapsulate
90      stmt.accept(new RlVisitor() {
91
92          public boolean visit(IPredname node) {
93
94              if (!node.inContextOf(RrlPackage.Literals.IN_MODULE) &&
95                      !node.inContextOf(RrlPackage.Literals.IMPORT_AS)) {
96
97                  if (node.inContextOf(RrlPackage.Literals.OUT_INTERFACE)) {
98                      IPredname pred =
99                          IPrednameImpl.load(node.toString() + sep +
100                             names.get(name.toString()) + sep +
101                             interfaceOut.toString());
102                     node.bind(pred);
103                 } else {
104                     IPredname pred =
105                         IPrednameImpl.load(node.toString() + sep +
106                             names.get(name.toString()) + sep +
107                             interfacePrivate.toString());
108                     node.bind(pred);
109                 }
110             }
111             return true;
112         }
113     });
114
115     // replace passive syntax
116     stmt.accept(new RlVisitor() {
117
118         public boolean visit(IOutInterface node) {
119             IHead head = new IHeadImpl();
120             node.accept(new RlVisitor(head) {
121
122                 public boolean visit(IHead node) {
123                     getParamFragment().bind(node);
124                     return true;
125                 }
126             });
127
128             if (head.isLoaded()) {
129                 node.bind(head);
130             }
131             return true;
132         }
133     });
134     return stmt;
135 }
136
137 /**
138  * IN-MODULE COMPOSER
139  *
140  */
141 @ReusewairComposer("InModule")
142 public static IAtom inModule(final IPredname module, final IAtom atom) {
143
144     atom.accept(new RlVisitor() {
```

```
145
146             public boolean visit(IPredname node) {
147                 // check if we are given information from the Import operator
148                 if (names.containsKey(module.toString()))
149                     node.bind(IPrednameImpl.load(node.toString() + sep +
150                             names.get(module.toString()) + sep +
151                             interfaceOut.toString()));
152                 // default
153                 else {
154                     System.err.println("The module name '" + module.toString() + "'
                            has not been declared");
155                     node.bind(IPrednameImpl.load(node.toString() + "_err_" +
156                             interfaceOut.toString()));
157                 }
158                 return true;
159             }
160         });
161         return atom;
162     }
163 }
```

LISTING 4.8: *Composition operators specified to properly transform fragments during composition of rule modules.*

Part III

Applications / Evaluation

5

Query Components: Modules for Xcerpt

[*This chapter is closely based on [6], but extended with more detailed explanations, further developed examples, parameterized modules and specification of the composition system implementing the module system.*]

As the amount and diversity of data available on the Web is constantly increasing, querying this great abundance of information is becoming more and more important. In fact, it is becoming less important to possess certain knowledge, but more important to know how to acquire it—know how to formulate a precise *query* to find the desired information. Query languages for different purposes are emerging in multitude. The survey in [10] mentions some existing query and transformation languages for Web and Semantic Web data, identifying 14 textual XML query languages and 24 for RDF [55] metadata.

Yet, many of these languages provide very little support to cope with the dramatic increase in information size and diversity. Increasing information *diversity*—data modeled according to standard, non-standard or exotic data schemata—results in increase of query size and complexity, which can weigh down even experienced query programmers. It must be easy for users to partition (both conceptually and from an evaluation point of view) query programs and to make such partitioning flexible enough to allow for reuse in different contexts. This is not the case unless the query language provides some means to separate large and complex query programs into smaller, properly isolated, and reusable fragments—*modules*. If we in addition provide usable interfaces to such modules, they allow for *separation of concern* of query programs via standard means of encapsulation. That is, we can hide the detail of how the modules are realized and instead have programmers rely on their interfaces.

Modules and their interfaces allow to "localize" the effect of the introduction of additional data sources or query tasks in query programs. By localizing a query task and moulding it into a module we essentially create a query *service* that can be reused across query programs and applications. There are different kinds of services that can be useful to query programmers and Web applications. For example, one part of a Web application is often concerned with extracting data from a set of sources, such as a set of Web pages, and possibly syndicating that data into a common view and format. Based on this syndicated data some transformations could be done, or new implicit data could be derived. Finally, the resulting data set should be generated into an appropriate for-

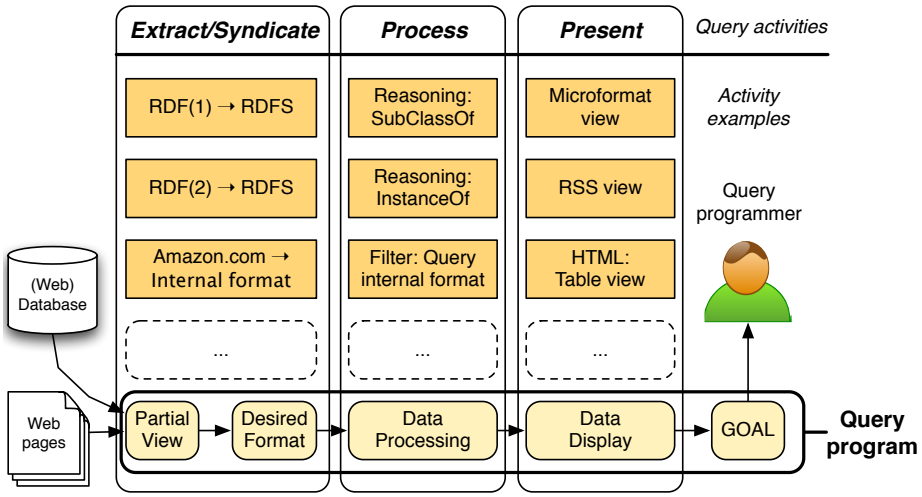


FIGURE 5.1: *In general, querying covers certain activities, each for which reusable entities (modules) can be created.*

mat, for example, by being generated into HTML for display in a Web browser. These different activities, illustrated in Figure 5.1, have to do with different *concerns* of the overall query application realization, such as data extraction (*Extract/Syndicate*), data management (*Process*) and data generation (*Present*). Without the possibility of using modules, each of these concerns have to be coded into a monolithic query program. Not only does this hamper reuse, but such a program can be very hard to maintain since a change in some part invariably affects some other part. It is valuable to localize and encapsulate knowledge about queries to minimize the impact of changes.

The main contribution of this chapter is *Modular Xcerpt*, a component-oriented extension of the rule-based query language Xcerpt [77]. The extension is focused on the *module* concept and allow programmers to define and deploy modules in their query programs. The contribution consists of two parts:

1. *Composition framework validation.* We show how the module extension to Xcerpt can be realized by instantiating the composition framework detailed in Chapters 2–4. The following issues are involved in this instantiation:

   (a) *Embedded Invasive Software Composition.* The module extension of Xcerpt is realized through Embedded ISC (E-ISC, cf. Chapter 4). This means that we leverage the ideas and techniques introduced in Chapters 2–4, and apply them for a particular language, Xcerpt. This gives us an opportunity to evaluate the composition framework. E-ISC allows to provide intuitive constructs for the purpose of a software composition approach. For Modular Xcerpt, this means intuitive constructs for defining and deploying modules. The benefit is that programmers already familiar with Xcerpt have little new to learn, which holds promise for fast adoption.

   (b) *Abstraction realization via composition.* We define a composition system for Modular Xcerpt. This includes defining the language extension, as well as defining the composition operators that implement the semantics of the

extension. That is, we use our composition framework to specify the realization of the module abstraction. Since the composition framework requires that programs of the extended language (Modular Xcerpt) are composed into equivalent programs of the base language (Xcerpt), we must explain how this composition is done. We introduce the notion of "stores" to ensure separation of modules in composed results, and hence to ensure a correct realization of the module abstraction.

2. *Modular Web querying.* We demonstration how Xcerpt can benefit from the module concept, both by partitioning query programs into logical reuse units, and as a consequence, by partitioning the execution of programs. This is demonstrated by several examples of how modules are defined, and how they are used in query programs.

This chapter is organized around these contributions. Following a brief introduction to Xcerpt, in Section 5.2 we discuss the benefits of modules by considering different use-cases and query scenarios. In Section 5.3 we then introduce the constructs we need to realize the use-cases and similar cases. In Section 5.4 we present a set of examples involving modules. In Section 5.5 we discuss how separation of modules can be ensured in composition results by introducing the notion of "stores." Section 5.6 then defines a concrete composition system that realizes the module extension to Xcerpt. Section 5.7 discusses related work and Section 5.8 concludes the chapter.

5.1 Background: Web query language Xcerpt

Xcerpt is a rule-based language for querying semi-structured data, for example XML or RDF (which has an XML serialization). The language follows, or is closely related to, the Logic Programming (LP) paradigm (see, for example, [68] for an introduction to LP). There are many publications on Xcerpt (see, e.g., [21, 22, 77]). Here we recall the basic constructs needed to understand our module extension. An Xcerpt *program* consists of a finite set of Xcerpt *rules*. The rules of a program are used to define data, or to derive new data from existing data (i.e. the data being queried). In Xcerpt, two different kinds of rules are distinguished: *construct rules* and *goal rules*. Their syntax are given in Listings 5.1 and 5.2, respectively, where anything enclosed between angle brackets (< and >) will be explained later. We will simply refer to (Xcerpt) *rules* when we do not distinguish between the two kinds of rules.

Construct rules are used to produce intermediate results while goal rules make up the output of programs. Rules have a *head* and optionally a *body*. Intuitively, rules are to be read: if *body* holds, then *head* holds. A rule lacking a *body* is interpreted as a *fact*, that is, the rule *head* always holds.

```
1 CONSTRUCT
2       <head>
3 FROM
4       <body>
5 END
```

LISTING 5.1: *A* construct rule.

```
1 GOAL
2       <head>
3 FROM
4       <body>
5 END
```

LISTING 5.2: *A* goal rule.

```
GOAL
       authors [ all author [ var X ] ]
FROM
       book [[ author [ var X ] ]]
END

CONSTRUCT
       book [ title [ "White Mughals" ], author [ "William Dalrymple" ] ]
END

CONSTRUCT
       book [ title [ "Stanley" ], author [ "Tim Jeal" ] ]
END
```

LISTING 5.3: *The construct rules define data about books and their authors and the goal rule queries this data for authors.*

While Xcerpt works directly on XML data, it also has its own data format. Xcerpt *data terms* model XML data and there is a one-to-one correspondence between the two notions. While XML uses labeled "tags," Xcerpt data terms use a square bracket notation. The data term `book [ title [ "White Mughals" ] ]`, for example, corresponds to the `<book><title>White Mughals</title></book>` XML snippet. The data term syntax provides a more readable XML syntax to use in queries.

Formally, the *head* of a rule is a *construct term* and the *body* is a *query*. A query is a set of *query terms* joined by some logical connective (e.g. `or` or `and`). Query terms are used for querying data terms and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique to match data terms.[1] Query terms can be configured to take partialness and/or ordering of the underlying data terms into account during matching. Square brackets are used in query terms when order is of importance, otherwise curly brackets may be used. E.g. the query term `a [ b [], c [] ]` matches the data term `a [ b [], c [] ]` while the query term `a [ c [], b [] ]` does not. However, the query term `a { c [], b [] }` matches `a [ b [], c [] ]` since ordering is said to be of no importance in the query term. Partialness of a query term can be expressed by using double, instead of single, brackets (i.e. `[[ ... ]]` or `{{ ... }}`). Query terms may also contain logical variables (denoted by capitalized identifiers preceded by keyword `var`, for example, `var X`). If so, successful matching with data terms results in variable bindings used by rules for deriving new data terms. For example, matching the query term `book [ title [ var X ] ]` with the XML snippet above results in the variable binding `{X / "White Mughals"}`. *Construct terms* are essentially data terms with variables. The variable bindings produced by queries in the body of a rule can be applied to the construct term in the head of the rule in order to derive new data terms. In the rule head, construct terms including a variable can be prefixed with the keyword `all` to group the possible variable bindings around the specific variable.

An example Xcerpt program relating to books is shown in Listing 5.3. The last two rules are facts and define two books, each with a title and an author. The first rule—a goal rule—defines the output of the program. It queries authors of books, and constructs a list of all found authors. The program in Listing 5.3 would result in the following data term as output:

[1] This technique is called *simulation unification*, please consult [78] for details.

```
GOAL
       <head>
FROM
       in { resource { "file:db.xml", "xml" },
            <query>
       }
END
```

LISTING 5.4: *A program with a single rule querying an external resource.*

```
authors [ author [ "William Dalrymple" ], author [ "Tim Jeal" ] ]
```

Both authors are in the answer because of the grouping construct (`all`) used in the construct term of the goal rule. Furthermore, the query in the goal rule matches the two facts by not considering the book titles since the partialness construct is used (`[[ ... ]]`).

A rule can also query an external resource, for example, a Web page or an XML database stored as a file. An example in given in Listing 5.4 where the XML file `file:db.xml` is being queried by a not further detailed query (<*query*>). The construct term of the rule is also omitted (<*head*>).

5.2 Use cases—Modular Querying

We will now look at the details of two query scenarios where the encapsulation of query tasks in modules is helpful. The constructs needed to realize these studied query scenarios will be introduced in the next section. While we will be more precise about what we mean by an Xcerpt module in subsequent sections, we shall here simply assume that a module is the encapsulation of a query task with appropriate interfaces to make use of the service.

The first scenario, presented in Section 5.2.1, deals mainly with encapsulation of schema information. That is, how localization of data schemata is helpful for managing query programs. For the second scenario in Section 5.2.2 we discuss more generic query services that can be encapsulated as modules.

5.2.1 Encapsulating and reusing schema information

The illustration in Figure 5.2 shows the three query activities mentioned above (*Extract/Syndicate*, *Process* and *Present*). This scenario mainly focuses on using modules for the purpose of simplified extraction of data and for presenting the result of a query (first and third activities). Any XML querying deals with data schemata at some level, but these two query activities are especially fragile since they deal with data structures and formats that are either i) dictated by a third-party, for example a Web page beyond the control of the query programmer, or ii) dictated by certain output devices or platforms, for example the rendering format of a mobile device.

1. *Schema querying.* Assume a set of query programs part of an online Web service/application that query a Web site for information. (It could also be a Web service providing XML-serialized output.) For sake of familiarity, let's assume it is the Amazon.com Web site (`http://www.amazon.com`). With non-modular

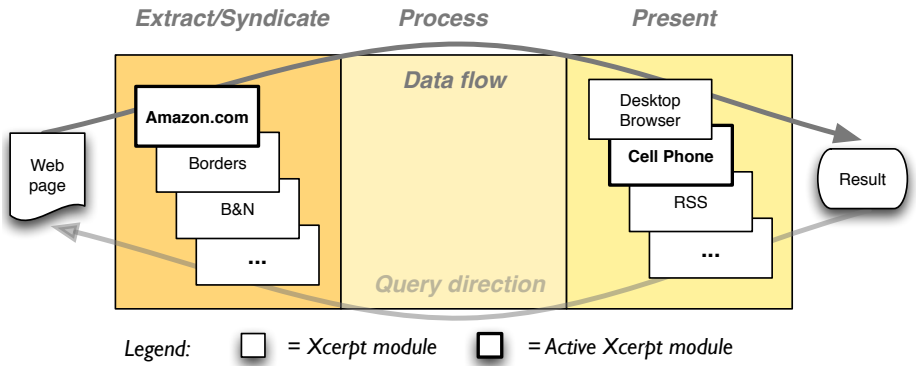


FIGURE 5.2: *The encapsulation of data schemata is beneficial since it allows for reuse and can localize the effect of schemata change.*

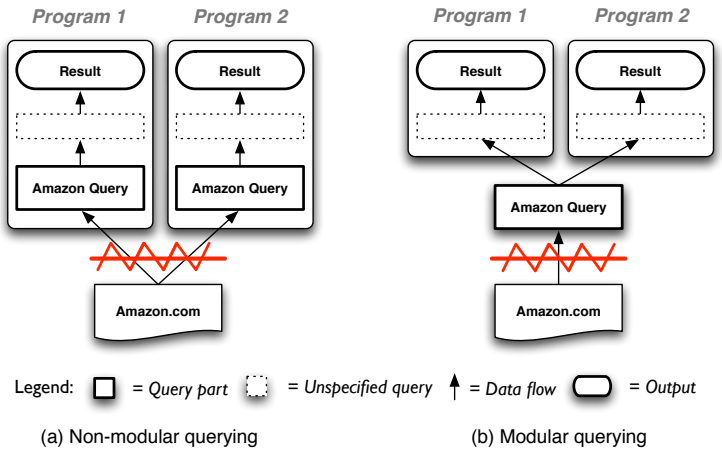


FIGURE 5.3: *The cost of maintaining query programs is reduced by encapsulating the details of how to access certain data (done in* (b)*, but not in* (a)*).*

querying, the structure of the Amazon.com Web site has to be encoded in every program wishing to query the site. Should those queries fail due to a change in the structure of the Web site (illustrated by the zig-zagged line in Figure 5.3), every program querying the site has to be modified and fixed, which leads to costly and difficult maintenance. If instead the queries to Amazon.com could be localized to a reusable query component (module) we would benefit from reuse and easier maintenance. This is illustrated in Figure 5.3 (b) by having a single and reusable *Amazon Query* component instead of having such queries encoded directly into each query program (as in Figure 5.3 (a))

2. *Schema provision.* Not only is it beneficial to encapsulate how data is accessed, as was discussed above, but often the same data set has to be displayed differently due to technical issues such as display devices, or because of social factors such as the target audience etc. When displaying queried data in a Web browser on a

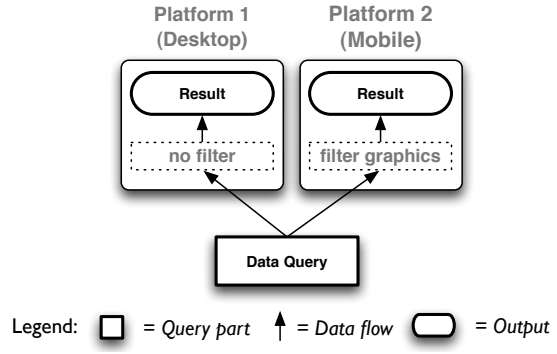


FIGURE 5.4: *Modules can be useful to encapsulate how certain data should be displayed.*

desktop computer, all of the data can possibly be displayed. However, when the same data should be displayed on a smaller device, for example a mobile phone, certain data, e.g. large graphics, might have to be filtered out.

Being able to quickly and effortlessly switch between those output formats and styles, based on the same data query, can be very helpful. This can be achieved by specifying the different filtering mechanisms in separate *provided* query programs (can also be seen as modules), illustrated in Figure 5.4, and encapsulating the data query as a module (*Data Query* in Figure 5.4).

The query scenarios discussed above have mainly been about separating and modularizing how data is either queried or presented as a final result. Thus:

QR1 It must be possible to define and encapsulate modules—thus localizing schemata information (both for querying and outputting)—such that they can be reused across query programs.

In both cases discussed above, either the *input* or the *output* data is fixed wrt. some external format that is not directly controllable or changeable by the query programmer. We can also consider more generic *transformational* query modules that are not connected to external data formats, but given certain input data, as dictated by the module programmer, they provide the services they implement as output. We shall consider the possible use of such modules in the following section.

5.2.2 Encapsulating and reusing data processing services

Within a query program itself—regardless of external data schemata—certain transformations of intermediate results may be required. Certain such transformations can be likened to *services* and be generalized and usable in many different query applications. An example of such a transformation service is a simple ontology reasoning service. A common purpose of an ontology is to arrange the central concepts of a modeled domain in a taxonomy (class hierarchy) using *subclass-of* relationships. Ontology reasoners can then be employed to infer any implicit *subclass-of* relationships.

A class hierarchy can be specified using a simple ontology language, for example RDF(S) [18]. Since RDF(S) ontologies have XML-serializations (RDF/XML), a query

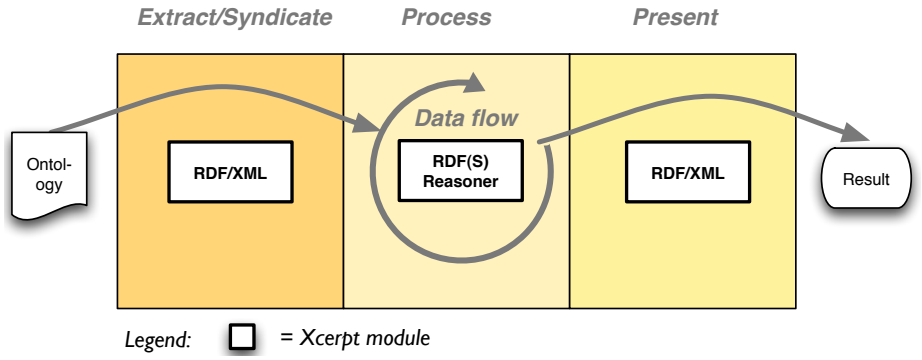


FIGURE 5.5: *Generic query tasks, such as a simple ontology reasoner (e.g. an RDF(S) document serialized in RDF/XML), can be valuable services that should be reusable across applications.*

program could access the explicit information contained in such an ontology *document* and compute the implicit *subclass-of* relationships [77, pp. 126–128]. Such computations can be formulated as a query task and be made into a module such that it can be reused across applications in need of that service. The input for the module (service) would be a list of explicit *subclass-of* relationships, and the output would be the same list but extended with any derived implicit *subclass-of* relationships. This query service idea is illustrated in Figure 5.5. Other such transformation services could be imagined. Thus, as a second query requirement, we have:

QR2 It must be possible to define generic and reusable data transformation services that, given a certain input, provides the expected output.

In the next section we shall discuss the language constructs needed for Xcerpt to support and realize the above-mentioned query scenarios wrt. modularization and reuse.

5.3 Requirements and constructs for Modular Xcerpt

The previous section discussed how modular querying can be useful and how separating query tasks into reusable modules makes life easier for Web query programmers. The query language Xcerpt does not, however, at the time of writing, support a module concept.

The only programming abstraction provided by Xcerpt is the *rule*. A rule can query an external resource or data constructed by another rule. An Xcerpt program is thus a set of rules with certain implicit dependencies. The programmer has the freedom of splitting the overall query task into any number of rules. During evaluation of a query program the same rule can be used several times and is in this sense reused for that particular query evaluation. Xcerpt does not, however, provide a way of reusing rules across programs. Nor does Xcerpt provide means to reuse larger query tasks (sets of collaborating and related rules). For reuse of rules or query tasks, we introduce the notion of *Xcerpt modules*.

Definition 5.1. *(*Xcerpt module (*Informal*)*) An* Xcerpt module *is a set of rules that can be imported and reused across programs. A module defines* interfaces *dictating how the module may successfully be used.*

The interfaces are defined by adorning construct terms or queries of the module's rules. Adorned query terms are part of the *required* interface and adorned construct terms are part of the *provided* interface. Modules can contain both construct and goal rules, but construct terms of goal rules cannot be part of module interfaces since goal rules only result in program output.

Definition 5.2. *(*Xcerpt module (*Formal*)*) Let Q represent a query, C a construct term in a construct rule, and G a construct term in a goal rule. We denote $C \leftarrow Q$ a construct rule, and $G \leftarrow Q$ a goal rule. Then the following is an* Xcerpt module *consisting of n rules:*

$$\widehat{C_1} \leftarrow Q_1, \ldots, G_k \leftarrow Q_k, \ldots, C_n \leftarrow \widehat{Q_n}$$

where each C_i or Q_j adorned with a $\widehat{\ }$ (hat) is part of the module interface. The following properties hold for a module: (i) No Q_i or $\widehat{Q_j}$ will match any $\widehat{C_k}$, and (ii) No $\widehat{Q_i}$ will match any C_j or $\widehat{C_k}$. That is, no rule in the module depends on a rule with an adorned construct term, and adorned queries can only match rules outside of the module.

In general a module can have several input and output interfaces. A module with at most one input and one output interface—at most one adorned query and one adorned construct term—seems to be a particularly common and useful case. Most of our examples will have one output interface, and possibly one input interface. Below we define and discuss concrete constructs needed to define modules and for making use of them in programs. It should be noted that it is also possible for modules to make use of other modules, called module *nesting*.

1. **Defining modules – constructs for module programmers.** Module programmers need constructs for defining sets of rules as modules and ways of declaring their interfaces.

   (a) *Module definition.* We can group sets of rules into modules and give such a set a mnemonic identifier using the *module* construct.

   ⟨*module*⟩ ::= `MODULE` ⟨*module-id*⟩ ⟨*import*⟩* ⟨*rule*⟩*

   The ⟨*module-id*⟩ construct is a simple string identifier, the ⟨*import*⟩ construct is defined below and the ⟨*rule*⟩ construct is the rule construct of Xcerpt. The *import* constructs inform us that a module can in turn import any number of other modules. The *module* construct is assumed, along with the *program*, to be a fundamental *unit* formulable by programmers.

   (b) *Module interfaces.* A module is considered to have a *required* interface if any of its rules are meant to query data produced by rules outside of the module. This can be allowed by adorning a top-level query with the `public` keyword.

   ⟨*interface-in*⟩ ::= `public` ⟨*top-level-query*⟩

The ⟨*top-level-query*⟩ construct is defined in Xcerpt and represents a query that is either a query contained directly in the rule body, or the top-most query term inside a complex query (conjunction or disjunction). Similarly, a module will require an *provided* interface if the data produced by the module is intended to be further processed. To achieve this, the `public` keyword may adorn a top-level construct term.

⟨*interface-out*⟩ ::= `public` ⟨*top-level-construct-term*⟩

The ⟨*top-level-construct-term*⟩ construct is again defined by Xcerpt, and is a construct term directly contained in a rule head. Both the ⟨*interface-in*⟩ and the ⟨*interface-out*⟩ constructs are assumed to be valid alternatives for the constructs they encompass. That is, where a ⟨*top-level-query*⟩ can be programmed, an ⟨*interface-in*⟩ construct can be placed. The equivalent holds for ⟨*interface-out*⟩.

Thus, a module programmer defines a set of rules, gives them a suitable name, and possibly defines the input and output interfaces of the module, all depending on the programmer's intension with the module.

2. **Deploying modules – constructs for module users.** Module users need to be able to (a) declare which modules they want to use in a program, to (b) query those declared modules, and to (c) provide data to the same modules, if required.

   (a) *Module import.* We can import modules into other modules or programs. This is done using the `IMPORT-AS` construct, defined by:

   ⟨*import*⟩ ::= `IMPORT` ⟨*module-ref*⟩ `AS` ⟨*alias-id*⟩

   The ⟨*module-ref*⟩ is the location or unique identifier of the module, while the ⟨*alias-id*⟩ is a string identifier. The ⟨*alias-id*⟩ can be used in the same program to refer to the declared module. The `IMPORT-AS` construct can be used before the rules of the module (or program) being defined.

   (b) *Module querying.* We can query the data produced by a module using the `IN-MODULE` construct:

   ⟨*in-module*⟩ ::= `IN` ⟨*alias-id*⟩ `(` ⟨*query*⟩ `)`

   The ⟨*alias-id*⟩ construct represents the precise module to query and the ⟨*query*⟩ represents the actual Xcerpt query. The query can only match against data produced by *provided* interfaces of the referred module. The `IN-MODULE` construct can be used where an Xcerpt ⟨*query*⟩ construct is allowed.

   (c) *Module provision.* We can feed, or provision, data to a module using the `TO-MODULE` construct:

   ⟨*to-module*⟩ ::= `TO` ⟨*alias-id*⟩ `(` ⟨*top-level-construct-term*⟩ `)`

   The ⟨*alias-id*⟩ construct represents the precise module to feed data into. The data produced by the `TO-MODULE` construct can only be matched by rules in the referred module that are part of its *required* interface, that is, rules with they keyword `public` used in its body. The `TO-MODULE` construct can be used where *top-level-construct-term*s are allowed.

Below we present a simple example making use of the above introduced constructs, and briefly study the consequences in terms of module encapsulation.

```
MODULE participants
IMPORT file:student.mx AS stud

CONSTRUCT
  public
    participants [
      all name [ var N ]
    ]
FROM
  IN stud (
    students [[
      name [ var N ]
    ]]
  )
END
```

LISTING 5.5: *Module* `A`*: Participants module in file* `file:particip.mx`.

```
MODULE student

CONSTRUCT
  public students [
    name [ "John Rowlands" ],
    name [ "Henry Stanley" ],
    name [ "Edmund Morel" ],
    name [ "Roger Casement" ]
  ]
END

CONSTRUCT
  students [
    name [ "William Sheppard" ] ]
END
```

LISTING 5.6: *Module* `B`*: Student data module in file* `file:student.mx`.

```
IMPORT file:particip.mx AS part

GOAL
  results [ all name [ var Name ] ]
FROM
  IN part (
    participants [[
      name [ var Name ] ]]
  )
END
```

LISTING 5.7: *Program* `P`*: The main query program.*

```
results [
  name [
    "John Rowlands" ],
  name [
    "Henry Stanley" ],
  name [
    "Edmund Morel" ],
  name [
    "Roger Casement" ]
]
```

LISTING 5.8: *The result of executing the query program* `P`.

Example 5.1. (*Simple Xcerpt modules and their usage*) This example deals with two modules and a main program. Module `A` (Listing 5.5) imports module `B` (Listing 5.6) and is itself imported into the main program `P` (Listing 5.7). We thus have the following dependency between the modules and the program (where $\longrightarrow$ denotes the dependency relation):

$$P \longrightarrow A \longrightarrow B$$

Module `B` defines data about students, their names in particular. Some of the data is declared to be part of the module interface, namely, where the construct term is adorned with the `public` keyword. Module `A` imports module `B` and queries it for student names using the `IN-MODULE` construct. Furthermore, module `A` "exports" the matched names, but in a different format. Again, this is the case since the construct term is adorned with the `public` keyword. The result of executing the main query program `P` is shown in Listing 5.8 (in Xcerpt's internal data term format).

The simple modules and query program in this example essentially passes the *public* data declared in module `B` into the main program `P`, via module `A`, as can be seen in the query result in Listing 5.8. Notice that the name `"William Sheppard"` is not part of the result since this data is not declared to be part of the interface of module `B`.

```
IMPORT file:student.mx AS stud

GOAL
  access_allowed []
FROM
  IN stud (
    students [[
      name [ "William Sheppard" ] ]]
  )
END
```

LISTING 5.9: *Failing to query module* `B`.

```
IMPORT file:student.mx AS stud

GOAL
  intrusion_achieved []
FROM
  students [[
    name [
      "Roger Casement" ]
  ]]
END
```

LISTING 5.10: *Failing to query module* `B`.

The programs in Listings 5.9 and 5.10 are constructed to test the encapsulation capabilities of the module system. Both the programs in Listings 5.9 and 5.10 return `<error>no results</error>` (empty results), but for different reasons. The program in Listing 5.9 correctly uses the `IN-MODULE` construct, but queries data that is not part of the interface of the imported module (cf. module in Listing 5.6). The program in Listing 5.10 queries data that is "visible" wrt. the imported module, but fails to actually query the imported module using the provided `IN-MODULE` construct. Hence, both queries return empty answers.

■

5.4 Examples: Modular Xcerpt

This section contains two main Modular Xcerpt examples. The first deals with modules for ontology reasoning. The second deals with a small query application for presenting music album information in different ways.

5.4.1 Ontology reasoning

In this section we show some Xcerpt modules that can be used for simple ontology reasoning. A common reasoning task is to derive implicit subclass-of relationships. The module in Listing 5.11 has been designed for this purpose, and is assumed to exist in file `file:rdfssubclassof.rxcerpt`. Given a set of explicit subclass-of relationships, it derives all the implicit ones. Note that this reasoning module calculates the transitive closure of some relationship and is in this sense general. However, to stay with the problem domain we use the special subclass-of relationship here. The actual calculation is of course done by Xcerpt, but what our module extension allows us to do is to define a set of reusable rules as a logical and reusable unit with interfaces.

The module consists of four rules. Informally they say the following, in order of definition:

1. Data terms related via `subclassof-deriv` are passed as the module output (**`public`**), consumable by modules or programs using this module.

2. Every class is its own subclass.

```
MODULE subclassOf

CONSTRUCT public output [ all subclassof [ var Sub, var Sup ] ]
FROM       subclassof-deriv [ var Sub, var Sup ]
END

CONSTRUCT subclassof-deriv [ var Cls, var Cls ]
FROM       or {  declsubclassof [ var Z, var Cls ],
                 declsubclassof [ var Cls, var Z ] }
END

CONSTRUCT subclassof-deriv [ var Sub, var Sup ]
FROM       or {   declsubclassof [ var Sub, var Sup ],
                        and {   declsubclassof [ var Sub, var Z ],
                                subclassof-deriv [ var Z, var Sup ]
                        }
           }
END

CONSTRUCT  declsubclassof [ var Sub, var Sup ]
FROM        public input [[ subclassof [ var Sub, var Sup ] ]]
END
```

LISTING 5.11: *Xcerpt module for deriving implicit subclass-of relationships from explicitly given ones. The module is assumed to be defined in file* `file:rdfssubclassof.rxcerpt`.

3. Derived subclass-of relationships are either: declared subclass-of relationships (`declsubclassof`), or (recursively) calculated as the transitive closure of the `subclassof-deriv` relationship, starting from a declared subclass-of relationship.

4. Subclass-of relationships given as input (**`public`**) are declared subclass-of relationships (`declsubclassof`).

The program in Listing 5.12 imports the module defined in Listing 5.11 and consists of two rules that say the following (in order of definition):

1. Give as output of the program all subclasses of the class named `"Vehicle"`. The imported module is queried (using **`IN`**) to get all the subclasses, including implicitly specified ones.

2. Query the OWL [73] file in `file:data/vehicles.owl` for explicitly declared subclass-of relationships, and give them as input to the imported reasoning module (using **`TO`**).

Let us assume that the file `file:data/vehicles.owl` contains the OWL ontology specified in Listing 5.13.[2] By composing the program in Listing 5.12 and executing

[2]This is not a standard OWL [73] serialization, but a simplified version. This is not only done for simplicity reasons, but due to limitations in the current Xcerpt prototype.

```
IMPORT file:rdfssubclassof.rxcerpt AS rdfs

GOAL   vehicles [ all var Sub ]
FROM   IN rdfs ( output [[ subclassof [ var Sub, "Vehicle" ] ]] )
END

CONSTRUCT
       TO rdfs ( input [ subclassof [ var Sub, var Sup ] ] )
FROM
       in { resource { "file:data/vehicles.owl", "xml" },
              owl {{
                     class {{ attributes { id { var Sub } },
                            subclassof {{ attributes { about { var Sup } } }}
                     }}
              }}
       }
END
```

LISTING 5.12: *Program making use of an Xcerpt module to calculate implicit subclass-of relationships.*

```
<?xml version="1.0" ?>
<owl>
   <class id="Vehicle" />
   <class id="Twowheelers">   <subclassof about="Vehicle" />       </class>
   <class id="Fourwheelers">  <subclassof about="Vehicle" />       </class>
   <class id="Car">           <subclassof about="Fourwheelers" />  </class>
   <class id="Truck">         <subclassof about="Fourwheelers" />  </class>
   <class id="Motorcycle">    <subclassof about="Twowheelers" />   </class>
   <class id="Harley">        <subclassof about="Motorcycle" />    </class>
   <class id="Moped">         <subclassof about="Twowheelers" />   </class>
</owl>
```

LISTING 5.13: *An OWL class hierarchy in a simplified serialization format.*

```
MODULE owlschema

CONSTRUCT
       public output [ all subclassof [ var Sub, var Sup ] ]
FROM
       in { resource { << file >>, "xml" },
              owl {{
                     class {{ attributes { id { var Sub } },
                            subclassof {{ attributes { about { var Sup } } }}
                     }}
              }}
       }
END
```

LISTING 5.14: *Module querying a yet unspecified (« `file` ») external resource (OWL document) for subclass-of relationships. The module is assumed to exist in file* `file:owlflatsubclass.rxcerpt`.

the result, we obtain the following result, all subclasses of the class `"Vehicle"`:

```
vehicles [ "Vehicle", "Twowheelers", "Fourwheelers"  "Car",
           "Truck", "Motorcycle", "Moped", "Harley"
]
```

In the example above we showed how to reuse a simple ontology reasoning module. To make the module as general and reusable as possible, we did not include the OWL schema in the module. Instead we required the users of the module to first query the ontology document and then to "push" the needed information into the module using the `TO-MODULE` construct. Ideally, we would also be able to reuse the schema querying the OWL document. We could put the relevant rule(s) into a module and reuse that across programs. The problem with this is however that we would also have to include the external resource (i.e. the OWL file location) in the module. This is because Xcerpt does not provide for a means to separate the resource to query from the query itself (cf. Listing 5.12). Without a possibility to separate the two we would end up with a module with rather limited reusability, since it can only be used for querying one particular ontology.

Even though we did not mention this as a requirement in Section 5.3, we will here show how this issue can be address by using the notion of a *slot* (cf. Chapter 2). Consider the module in Listing 5.14 (in file `file:owlflatsubclass.rxcerpt`). It contains a single rule querying an unspecified (**`<<file>>`**) external resource (presumed to be an OWL document) for subclass-of relationships. In this case, the schema information contained in the module is truly reusable. Making use of the module in Listing 5.14, we can now rewrite the program in Listing 5.12 to the one in Listing 5.15.

In Listing 5.15 we import the module in Listing 5.14 and parameterize (**`WITH`**) it with the particular file we want to reason on (here: `"file:data/vehicles.owl"`). The result from executing the program is the same as before. So, in this example, we make use of two general and reusable modules to achieve our goal. The modules contain query parts that otherwise would have had to be included in a monolithic program, without reusable entities.

Let us continue to develop the current example. It can also be useful to query an OWL document for individuals (instances). The module in Listing 5.16 accomplishes this, and is assumed to exist in file `file:owlindividuals.rxcerpt`.

```
IMPORT file:rdfssubclassof.rxcerpt AS rdfs
IMPORT file:owlflatsubclass.rxcerpt AS owl
       WITH (file => "file:data/vehicles.owl")

GOAL  vehicles [ all var Sub ]
FROM  IN rdfs ( output [[ subclassof [ var Sub, "Vehicle" ] ]] )
END

CONSTRUCT  TO rdfs ( input [ subclassof [ var Sub, var Sup ] ] )
FROM       IN owl ( output [[ subclassof [ var Sub, var Sup ] ]] )
END
```

LISTING 5.15: *Further modularized program for finding all subclasses of "Vehicle."*

```
MODULE owlindividuals

CONSTRUCT
       public output [ all individual [ var Name, var Type ] ]
FROM
       in { resource { << file >>, "xml" },
              owl {{
                     individual {{ attributes { id { var Name } },
                            type {{ attributes { about { var Type } } }}
                     }}
              }}
       }
END
```

LISTING 5.16: *Module querying a yet unspecified (« `file` ») external resource (OWL document) for individuals. The module is assumed to exist in file `file:owlindividuals.rxcerpt`.*

```
IMPORT file:rdfssubclassof.rxcerpt AS rdfs
IMPORT file:owlflatsubclass.rxcerpt AS owl
       WITH (file => "file:data/vehicles-with-indiv.owl")
IMPORT file:owlindividuals.rxcerpt AS ind
       WITH (file => "file:data/vehicles-with-indiv.owl")

GOAL result [ all instances [ var Name, var Sup ] ]
FROM and { IN ind ( output [[ individual [ var Name, var Type ] ]] ),
           IN rdfs ( output [[ subclassof [ var Type, var Sup ] ]] ) }
END

CONSTRUCT  TO rdfs ( input [ subclassof [ var Sub, var Sup ] ] )
FROM       IN owl ( output [[ subclassof [ var Sub, var Sup ] ]] )
END
```

LISTING 5.17: *Query program deriving types of declared instances.*

```
<individual id="honda1500"> <type about="Motorcycle" />  </individual>
<individual id="volvoxc90"> <type about="Car" />         </individual>
```

LISTING 5.18: *Declaration of two OWL individuals:* `honda1500` *and* `volvoxc90`*.*

We can now write a query program that makes use of three modules: 1) one for deriving implicit subclass-of relationships, 2) one that extracts declared subclass-of relationships, and 3) one that extracts instance declarations. The program in Listing 5.17 derives implicit class memberships for declared instances. The construct rule provides data for the subclass-of reasoning module. The goal rule queries the result of the reasoning module, as well as the module querying the instance declarations.

Let us assume that the declarations in Listing 5.18 are added to the specification in Listing 5.13, assumed to exist in file `file:data/vehicles-with-indiv.owl`. Two individuals are declared: a motorcycle `honda1500`, and a car `volvoxc90`. Now, if we execute the composition result of Listing 5.17, we get the following result:

```
result [
  instances [     "honda1500",    "Motorcycle"   ],
  instances [     "honda1500",    "Twowheelers"  ],
  instances [     "honda1500",    "Vehicle"      ],
  instances [     "volvoxc90",    "Car"          ],
  instances [     "volvoxc90",    "Fourwheelers" ],
  instances [     "volvoxc90",    "Vehicle"      ]
]
```

That is, `honda1500` is a motorcycle, a two-wheeler, and a vehicle, while `volvoxc90` is a car, a four-wheeler, and a vehicle.

In this section we looked at how data processing services (OWL reasoning) and schema encapsulation can be useful for query programmers. In the next section we give another example of benefits with modular queries, but relating to querying and publishing Web page data.

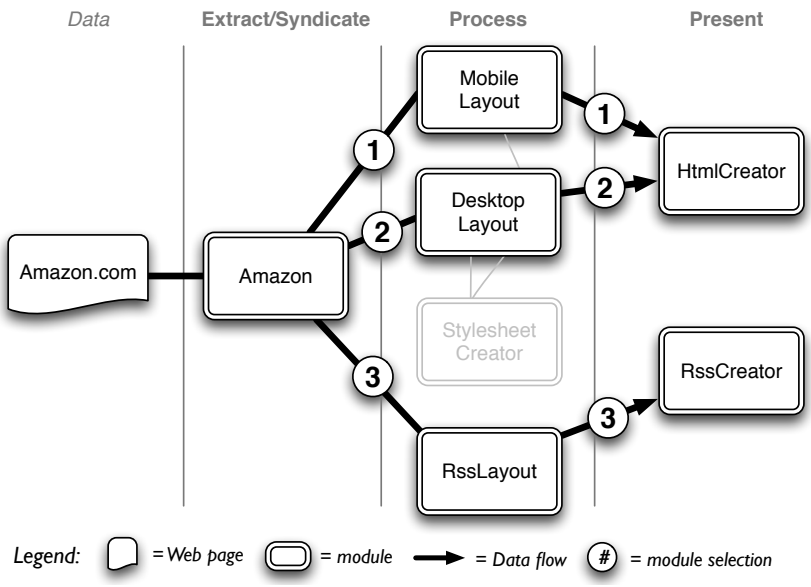


FIGURE 5.6: *Illustration of query modules for extracting Web data, transforming it in different ways, and generating the results in different formats.*

```
IMPORT file:import/Amazon.rxcerpt AS Amazon
IMPORT file:html/layout/MobileLayout.rxcerpt AS Layout
IMPORT file:html/HtmlCreator.rxcerpt AS Creator

GOAL  out { resource {"file:result.html", "xml"}, var Data }
FROM  IN Creator ( var Data )
END

CONSTRUCT
       TO Creator (
       creator [ title [ "Music Library" ], style [ "data/yellowstyle.css" ],
              data [ var Data ] ] )
FROM  IN Layout ( var Data )
END

CONSTRUCT TO Layout ( input [ all var Cdinfo ] )
FROM       IN Amazon ( var Cdinfo -> cd [[ ]] )
END

CONSTRUCT TO Amazon ( var Data )
FROM       in { resource { "file:data/pixel_revolt.html", "xml" }, var Data }
END

CONSTRUCT TO Amazon ( var Data )
FROM       in { resource { "file:data/the_letting_go.html", "xml" }, var Data }
END
```

LISTING 5.19: *Modular query program that queries HTML pages for music album information and displays the result according to modules* `Creator` *and* `Layout`.

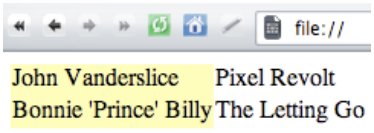


FIGURE 5.7: *Final output from the modular query program in Listing 5.19 (corresponding to path (1) in Figure 5.6). The display of the two music albums is in a format suitable for mobile devices.*

5.4.2 Web Music Library

The example presented in this section concerns a simple modular query application for extracting and presenting information about music. The query modules involved in the example are illustrated in Figure 5.6. There are modules for extracting information from Web pages (`Amazon`), modules for processing information (e.g. `MobileLayout` and `RssLayout`), as well as modules for presenting the final result data in some particular format (`HtmlCreator` and `RssCreator`). Each of the modules illustrated in Figure 5.6 can be found in Appendix 5.A.

In this example we will create three different end results, essentially by substituting certain modules for others. Each result will present information extracted from the Web about music albums, but in different ways. More precisely, the following:

1. In HTML, but intended for devices will small displays (e.g. a mobile device).
2. In HTML, intended for normal desktop browsers (i.e. for a large screen).
3. An RSS[3] file with an item for each music album.

The program for the first output (1) is shown in Listing 5.19. The program imports the three modules `Amazon`, `MobileLayout` and `HtmlCreator`. Two Web pages are queried that correspond to two albums. The Web pages are assumed to be Amazon Web pages, but are in this example local and simplified versions (hence the `file:` protocol). The content of the pages are sent to the `Amazon` module for extraction of the Amazon-specific data into an internal format (common between certain modules). The `Amazon` module is then queried and the data is passed to the layout module (here: `MobileLayout`). Then the processed data is passed to the `HtmlCreator` module to create a HTML document. As a final step, the result of the program is constructed and directed to the file `file:result.html`.

By first composing and executing the program in Listing 5.19, and then displaying the resulting file `file:result.html` in a Web browser, you would see something like Figure 5.7. This display format of the two albums is suitable for a smaller device, where the most important information is present.

What if we want to achieve the second output (2) from above? By simply replacing some of the used modules, we can get a different result. Let's replace the `MobileLayout` module with the `DesktopLayout` module. This means changing Line 2 in Listing 5.19 into:

```
IMPORT file:html/layout/DesktopLayout.rxcerpt AS Layout
```

[3]Real Simple Syndication. See, e.g., `http://cyber.law.harvard.edu/rss/rss.html`.

```
<rss>
  <channel>
    <title>Music Library</title>
    <item>
      <title>John Vanderslice - Pixel Revolt</title>
      <description>No description</description>
    </item>
    <item>
      <title>Bonnie 'Prince' Billy - The Letting Go</title>
      <description>No description</description>
    </item>
  </channel>
</rss>
```

LISTING 5.20: *RSS output file with music album information (corresponding to path (3) in Figure 5.6).*

If we re-compose the program, execute it again, and re-display the output file in the Web browser, we would see something like Figure 5.8. That is, the `DesktopLayout` module uses a different layout mechanisms and includes more information (such as album art), since this makes sense when displaying on a larger screen.

To achieve the third output (3) option discussion above, we again do not have to change much. We can simply replace the second and third import statement from Listing 5.19 by the following (and rename the output file to `file:result.rss`):

```
IMPORT file:rss/RssLayout.rxcerpt AS Layout
IMPORT file:rss/RssCreator.rxcerpt AS Creator
```

In this case, instead of generating a Web page displaying the album information, we end up with an RSS file as shown in Listing 5.20. The resulting RSS file here does not contain very much information, but more interesting data could naturally be included. The interesting thing here is that by changing remarkably small amounts of code, essentially replacing some query modules for others, we can completely change the result and map it to our current needs. This is the power of modules in rule-based Web query languages.

5.5 Composing Modular Xcerpt programs

Our goal is to realize the semantics of Modular Xcerpt programs (cf. Section 5.4) via composition. In Section 5.6 we develop a concrete composition system for this purpose, where we deploy the composition techniques presented in Chapters 2–4. The composition framework requires programs and modules of Modular Xcerpt to be composed into plain Xcerpt programs, while retaining the semantics of the original programs/modules. This enables the reuse the existing Xcerpt interpreter for Modular Xcerpt. Before we go into the details of composition systems, we present the composition strategy without discussing how it concretely is realized. In particular, we explain how module encapsulation can be retained in the composition result via the notion of "stores."

The purpose of the *store* is to ensure proper module encapsulation. A store can be likened to a virtual (XML) database associated with a unique identifier. Every module

FIGURE 5.8: *Final output from the modular query program in Listing 5.19 using the* `DesktopLayout` *module instead of the* `MobileLayout` *module (corresponding to path (2) in Figure 5.6).*

Section	Data stored in the section
`out`	*Data part of the* provided *interface – data to be queried using the* `IN-MODULE` *construct.*
`private`	*Data for the internal use of the module – not accessible from outside the module.*
`in`	*Data injected using the* `TO-MODULE` *construct and used by the* required *rules of the module.*

TABLE 5.1: *A store consists of three sections:* `out`, `private` *and* `in`.

is assumed to be associated with a store. A store is divided into three sections: `out`, `private` and `in`. See Table 5.1 for their explanations.

In order for a module to be properly encapsulated, every data term internally constructed by the module (not part of the module's interface) should only be usable by other rules of the same module. If this is not the case then the encapsulation of the module is violated. Or, if rules within the module can query rules outside the module that do not intentionally provide data to the module, then encapsulation is also violated. By directing any data terms constructed into the suitable section of the considered module's store, proper separation, and hence encapsulation, can be ensured.

The extended constructs (`IMPORT-AS`, `IN-MODULE`, `TO-MODULE`) that the module programmers make use of are responsible for directing rules to the correct section of the considered module's store.

– *Module import.* When importing a module using the `IMPORT-AS` construct, every top-level construct term and query term of each rule is directed to the appropriate section of the module's store. This is in general done by transforming each module rule in the following way:

```
CONSTRUCT
  <head>
FROM
  <body>
END
```

⟶

```
CONSTRUCT
  <STORE: <head> >
FROM
  <STORE: <body> >
END
```

where each construct enclosed within < and > is as of yet unspecified. In general the `<STORE: <term> >` construct can be expanded to the following:

```
store [ id [ <module-id> ], section [ <section> ], <term> ]
```

where `<module-id>` is a string containing a unique identifier of the module (a URI or the location of the module), `<section>` a string indicating which section of the store is being referred to (`out`, `private` or `in`), and `<term>` is the construct term or query term considered. We refer to a *term* when it is irrelevant if we mean a construct or a query term.

If the considered term is not part of the module interface, then the section used is `private`. If the term is a construct term adorned with the `public` keyword, and is hence part of the *provided* interface, then we use the `out` section of the store.

In the same manner, if we consider an adorned query, then we use the `in` section. The following is an example of a single rule with an adorned construct term:

```
CONSTRUCT
  public data_out [
     var X
  ]
FROM
  data [
    var X
  ]
END
```

⟶

```
CONSTRUCT
  store [ id [ <module-id> ],
          section [ "out" ],
          data_out [ var X ] ]
FROM
  store [ id [ <module-id> ],
          section [ "private" ],
          data [ var X ] ]
END
```

– *Module querying.* The `IN-MODULE` construct is provided for querying specific modules. Such queries are meant to query the data terms constructed by a module as part of its *provided* interface, and are hence referred to the `out` section of the referred module's store. The following transformation is done:

```
CONSTRUCT
  ...
FROM
  IN mod (
    <query>
  )
END
```

⟶

```
CONSTRUCT
  ...
FROM
  store [ id [ <module-id> ],
          section [ "out" ],
          <query> ]
END
```

– *Module provision.* The `TO-MODULE` construct can be used for providing data to be used by a module. For this data to be made available to the module, it needs to be directed to the `in` section of the module's store. This is done via the following transformation:

```
CONSTRUCT
  TO mod (
    <construct term>
  )
FROM
  ...
END
```

⟶

```
CONSTRUCT
  store [ id [ <module-id> ],
          section [ "in" ],
          <construct term> ]
FROM
  ...
END
```

Example 5.2. (*Simple module composition*) The following example makes use of a simple identity module, that is, a module that simply returns the data terms it receives as input (Listing 5.22). The identity module could be implemented using a single rule, but we use two rules to show how the internal communication (between the two rules) is directed to the `private` section of the module's store.

The program in Listing 5.21 makes use of the identity module by sending it some data, expecting to get the same output as result when querying the module. The result of the query program is, as expected:

```
result [ "value" ]
```

The query program and the module, written in the Modular Xcerpt language, are composed to the plain Xcerpt programs show in Listings 5.23 and 5.24, respectively.

Notice that all the constructs belonging to the extended language, Modular Xcerpt, have been removed in the composed results. The programs in Listings 5.23 and 5.24 can be merged into a single program and executed by the Xcerpt interpreter to yield

```
IMPORT file:ident.mx AS ident

GOAL
  result [
    var X
  ]
FROM
  IN ident ( ident_out [ var X ] )
END

CONSTRUCT
  TO ident ( ident_in [ "value" ] )
END
```

LISTING 5.21: *A query program using the identity module.*

```
MODULE identity_module

CONSTRUCT
  public ident_out [ var X ]
FROM
  internal [ var X ]
END

CONSTRUCT
  internal [ var X ]
FROM
  public ident_in [ var X ]
END
```

LISTING 5.22: *An identity module at* `file:ident.mx`.

```
GOAL
  result [
    var X
  ]
FROM
  store [
    id [ "file:ident.mx" ],
    section [ "out" ],
    ident_out [ var X ]
  ]
END

CONSTRUCT
  store [
    id [ "file:ident.mx" ],
    section [ "in" ],
    ident_in [ "value" ]
  ]
END
```

LISTING 5.23: *Program in Listing 5.21 composed to Xcerpt.*

```
CONSTRUCT
  store [ id [ "file:ident.mx" ],
          section [ "out" ],
          ident_out [ var X ] ]
FROM
  store [ id [ "file:ident.mx" ],
          section [ "private" ],
          internal [ var X ] ]
END

CONSTRUCT
  store [ id [ "file:ident.mx" ],
          section [ "private" ],
          internal [ var X ] ]
FROM
  store [ id [ "file:ident.mx" ],
          section [ "in" ],
          ident_in [ var X ] ]
END
```

LISTING 5.24: *Module in Listing 5.22 composed to Xcerpt.*

the expected result shown above. Thus, the composed and merged program is the realization of the abstractions used in Listings 5.21 and 5.22. The realization ensures that modules are encapsulated using the concept of stores.

■

Certain special cases are handled while composing modular query programs:

1. *External resources.* If a module queries an external resource, then the query is not transformed, since the query must match the format of that resource.

2. *Complex queries.* We remember that queries are not simply query terms, but sets of query terms joined by logical connectors, such as `or` or `and`. When transforming a conjunctive or disjunctive query, the transformations are done on the top-level of each involved query term. The following example illustrates the transformation:

```
CONSTRUCT
  ...
FROM
  or {
     <qt1>,
     and { <qt2>,
           <qt3> } }
END
```

⟶

```
CONSTRUCT
  ...
FROM
  or {
     <STORE: <qt1> >,
     and { <STORE: <qt2> >,
           <STORE: <qt3> > } }
END
```

   where `<qt1>`, `<qt2>` and `<qt3>` are query terms and `<STORE: ...>` is the appropriate store specification.

3. *Module nesting.* It is possible for modules to import other modules, so-called module nesting. During the transformation of a module, encountered `IN-MODULE` and `TO-MODULE` constructs are transformed wrt. the modules they are referring to.

5.5.1 Refined module encapsulation

The store concept described above ensures basic encapsulation capabilities for Modular Xcerpt and is attractive for its simplicity. However, there are certain situations where associating one store per module is not sufficient. Consider the situation where two modules (`A`, `B`) imports a third one (`C`) and both `A` and `B` injects data into the store associated with `C`. In such a case, after module `C` has processed the data, module `A` *may* receive data initially injected by module `B`. As such, modules `A` and `B` are not kept separate, which violates the encapsulation property of our module concept. A similar problem can also occur when the same module is imported more than once in the same program. This is not a limit of the store approach, but due to the assumption of the existence of one store per module.

To address this problem, we can associate stores not with a module, but with a module *import*. This can be seen as instantiating a store for each module import with a unique identifier. In our simple example we would thus end up with the two stores `C<#1>` and `C<#2>` for module `C`, due to two imports of it. The `#1` and `#2` represents the unique identifiers generated for the purpose of the separation of the involved stores.

5.6 Framework Evaluation: Composition System

In this section we specify a composition system for Modular Xcerpt by instantiating the framework described in Chapters 2 and 4. We recall the development process mentioned in Section 4.4.2 and go through the same development steps below.

D0 *Base grammar.* We first assume the existence of the grammar of the base language, in this case, Xcerpt's grammar. We do not present the complete grammar here, but provide the overall structure that is needed to understand the component model specification in the next step. Furthermore, we only specify the abstract syntax.

⟨*XcerptUnit*⟩ ::= ⟨*XcerptProgram*⟩
⟨*XcerptProgram*⟩ ::= ⟨*XcerptStatement*⟩+
⟨*XcerptStatement*⟩ ::= ⟨*GoalRule*⟩ | ⟨*ConstructRule*⟩
⟨*GoalRule*⟩ ::= ⟨*GoalQueryRule*⟩ | ⟨*GoalFact*⟩
⟨*ConstructRule*⟩ ::= ⟨*ConstructQueryRule*⟩ | ⟨*ConstructFact*⟩
⟨*GoalQueryRule*⟩ ::= ⟨*ConstructTerm*⟩ ⟨*QueryTerm*⟩
⟨*GoalFact*⟩ ::= ⟨*ConstructTerm*⟩
⟨*ConstructQueryRule*⟩ ::= ⟨*ConstructTerm*⟩ ⟨*QueryTerm*⟩
⟨*ConstructFact*⟩ ::= ⟨*ConstructTerm*⟩

The above has defined the main Xcerpt constructs, namely, the different kinds of rules. Next we define construct terms:

⟨*ConstructTerm*⟩ ::= ⟨*Identifier*⟩ | ⟨*OutResource*⟩ | ⟨*Variable*⟩ | ⟨*StructuredCt*⟩
⟨*OutResource*⟩ ::= ⟨*Resource*⟩ ⟨*Type*⟩ ⟨*ConstructTerm*⟩
⟨*StructuredCt*⟩ ::= ⟨*Identifier*⟩ ⟨*ChildrenListCt*⟩
⟨*ChildrenListCt*⟩ ::= ⟨*OrderedChildrenListCt*⟩ | ⟨*UnorderedChildrenListCt*⟩
⟨*OrderedChildrenListCt*⟩ ::= ⟨*ConstructTerm*⟩+
⟨*UnorderedChildrenListCt*⟩ ::= ⟨*ConstructTerm*⟩+
⟨*Variable*⟩ ::= ⟨*Name*⟩
⟨*Identifier*⟩ ::= STRING
⟨*Name*⟩ ::= STRING
⟨*Resource*⟩ ::= LOCATION
⟨*Type*⟩ ::= QUOTED_STRING

Query terms can be joined by logical connectors, we only show this aspect here:

⟨*QueryTerm*⟩ ::= ... | ⟨*ConditionQt*⟩
⟨*ConditionQt*⟩ ::= ⟨*ConjunctionQt*⟩ | ⟨*DisjunctionQt*⟩
⟨*ConjunctionQt*⟩ ::= ⟨*QueryTerm*⟩+
⟨*DisjunctionQt*⟩ ::= ⟨*QueryTerm*⟩+

The remaining part of the Xcerpt grammar is not needed to understand the following description, nor, in fact, to write the required composition operators. Interested readers can find full specifications in, for example, [20]. The above is a simplification of those specifications, but enough to create useful examples and demonstrate proof-of-concept.

D1–3 *Adapt base grammar to ISC, define domain-appropriate constructs and separate out active syntax.* The C_mSL^+ specification in Listing 5.25 extends the Xcerpt base grammar for the purpose of working with modules. The extension introduces intuitive constructs for defining and deploying modules, in line with the constructs defined in Section 5.3.

As can be seen, we introduce slots for three constructs, namely: `QueryTerm`, `ConstructTerm` and `Resource`.[4] The first two slot constructs are introduced to simplify the implementation of the composition operators, especially for handling encapsulation using stores (cf. Section 5.5). For example, a store can abstractly be defined as:

```
store [ id [ <module-id> ], section [ <section> ], <term> ]
```

The `<module-id>`, `<section>` and `<term>` parts are then replaced with appropriate values. In the implementation we will define the above abstract specification as a concrete internal fragment, and each of the unspecified parts as slots. Such an internal fragment will look something like:

```
store [ id [ «id» ], section [ «section» ], «term» ]
```

Such an internal fragment can then be used in the composition operator implementations to construct appropriate store terms. Slots help us to do this in an easy way. An example of how this is done is shown in Listing 5.26. Slots for the `Resource` construct are introduced due to our desire to properly encapsulate schema information (cf. Listing 5.14).

D4 *Generating component model.* Once the component model has been specified, it can be generated. This involves generating the core composition language—a Java API for Xcerpt composition—enabling us, for example, to define query terms using the Java type `IQueryTerm`. This also allows us to process query terms *within* other fragments, thus, to access them implicitly. The kind of fragments that will be available to a developer in this manner are the ones enumerated using the `fragtypes` construct in Listing 5.25.

Notice that all the passive syntax constructs are listed in the `fragtypes` statement. This is so because we need to work with them in order to compose them into constructs of the base language.

D5 *Defining composition operators for active syntax constructs.* The next step in the development process is to associate composition operators with the active syntax constructs introduced in the previous step. We notice that there are four such constructs defined by the nonterminals `ImportStatement`, `ImportWith`, `InModule` and `ToModule`. Hence, we must define composition operators for these constructs. The operator implementations can be found in the appendix of this chapter. Here we look at one of them in detail, and leave the study of the others to the reader. The operator definition in Listing 5.26 corresponds to the `IN-MODULE` construct.

Notice that the implementation method signature directly corresponds to the syntactical definition of the construct in the component model (cf. Listing 5.25).

[4] Resources are Xcerpt's external resources. That is, the specifications of which files to query.

```
extends file:xcerpt.gr @ x as file:mxcerpt.gr .

% slots constructs
slotify x.QueryTerm .
slotify x.ConstructTerm .
slotify x.Resource .

% passive syntax

ModuleDefinition  =  moduleName:x.Name, moduleStmt:x.XcerptStatement* .
ModuleDefinition  <> x.XcerptProgram .

OutInterface      =  interface:x.ConstructTerm .
OutInterface      <> x.ConstructTerm .

InInterface       =  interface:x.QueryTerm .
InInterface       <> x.QueryTerm .

% active syntax

% IMPORT <loc> AS <mod>
ImportStatement   =  moduleLocation:ModuleDefinition [ @Location ],
                     moduleName:x.Name .
ImportStatement   <> x.XcerptStatement .
ImportStatement   -> @Composer .

% IMPORT <loc> AS <mod> WITH ( <slot> => <value>, ...)
ImportWith        =  moduleLocation:ModuleDefinition [ @Location ],
                     moduleName:x.Name, slot:x.Name, value:x.Resource .
ImportWith        <> x.XcerptStatement .
ImportWith        -> @Composer .

% IN <mod> ( <query> )
InModule          =  moduleName:x.Name, interface:x.QueryTerm .
InModule          <> x.QueryTerm .
InModule          -> @Composer .

% TO <mod> ( <cterm> )
ToModule          =  moduleName:x.Name, interface:x.ConstructTerm .
ToModule          <> x.ConstructTerm .
ToModule          -> @Composer .

fragtypes { x.Name, x.XcerptStatement, x.QueryTerm, x.ConstructTerm, x.Resource,
            OutInterface, InInterface, ModuleDefinition }
```

LISTING 5.25: C_mSL^+ *specification for extending Xcerpt with a module concept.*

```
public static IQueryTerm inModule(IName name, IQueryTerm queryTerm) {

	// use internal fragment
	final IQueryTerm localQueryTerm =
		IQueryTermImpl.load("store [ id [ <<id>> ], section [ <<vis>> ],
			<<qTerm>> ]");
	// check if we are given information from the Import operator
	if (names.containsKey(name.toString())) {
		// get the name of the module (using a Hashtable 'names')
		final String id = "\"" + names.get(name.toString()) + "\"";
		// construct the internal fragment
		localQueryTerm.bind("id", IQueryTermImpl.load(id));
		localQueryTerm.bind("vis", IQueryTermImpl.load("visout[]"));
		localQueryTerm.bind("qTerm", queryTerm);
	} else {
		System.err.println("The module name '" + name.toString() +
			"' has not been declared");
	}
	// return and replace queryTerm with the local fragment
	return localQueryTerm;
}
```

LISTING 5.26: *Composition operator specified for the active syntax construct* `IN-MODULE`.

The implementation of this construct is quite simple. First, it defines an internal fragment of type `QueryTerm`, which looks like this:

```
store [ id [ «id» ], section [ «vis» ], «qTerm» ]
```

This is the query term that finally will replace the call of the operator. As can be seen, the internal fragment defines three slots, such that it can be parameterized with the proper values. The `id` slot should be bound to the identifier of the referenced module. This is achieved by looking up the module identifier in the hash table `names` (not defined, only used, in Listing 5.26). This hash table contains all alias-identifier pairs for all imported modules (if the alias is not contained in the hash table, an error message is given). Once the module identifier has been found, the slot `id` is bound with it. Then the appropriate visibility is associated with the internal fragment, which corresponds to the appropriate store section (cf. Section 5.5). For the `IN-MODULE` construct this is always the `out` section. Notice that in this concrete implementation we use a term instead of a string to indicate the store section. For example, `visout[]`, `visin[]` and `vispriv[]` instead of `"out"`, `"in"`, and `"private"`, respectively. Finally, the slot `qTerm` is bound with the query term operated on by the `IN-MODULE` construct (here `queryTerm` which is passed as an argument to the operator). As a last step, the transformed internal fragment is returned to replace the location where the operator was invoked.

5.7 Related Work

Though many rule languages for the Web fail to provide modules, this is not true for the two preeminent Web query languages, XSLT [24] and XQuery [13]. XSLT can be considered a rule language, however using precedence rather than union semantics

for multiple applicable rules. Rule precedence is also the dominating issue for XSLT's module system which provides intricate mechanisms for determining the precedence of rules from different modules. Nevertheless, the resulting module system is considerably less powerful (no scoped import, limited parameterization: apply-imports) yet needs a more complex semantics than module-free XSLT, quite in contrast to our approach.

It is worth mentioning that XQuery also provides a module system, however without parameterization, but as a function programming language requires explicit flow control in all cases. SPARQL [75], finally, the recently proposed RDF query language, has no concept of user defined program units (such as rules, functions, procedures, etc.) and thus no use for a module concept in the sense of our approach. However, rule-based extensions for SPARQL (in the spirit of Datalog) could certainly profit from the module system illustrated here using Xcerpt.

The arguably most comprehensive treatment of modules in logic programming is presented in [19]. It is far more expressive than our approach but at the price of a complex semantics and several operations with, in our opinion, little practical use (such as module intersection or renaming). We believe that a single well-designed union operation with clear interfaces together with a strong reliance on views as a core feature of rule languages is not only easier to grasp but also easier to realize.

5.8 Summary

In this chapter we have presented and discussed Modular Xcerpt, an extension of the Web query language Xcerpt with modules. We argue that one way of coping with the diversity of information on the (Semantic) Web is *modular* query authoring and execution. We have shown, through several examples, how modules can help query programmers in better understanding their programs (by separating out parts into self-sustainable units, that is, through support for separation of concern), while at the same time enable reuse of already developed functionality.

The realization of the module concept was achieved through composition. We employed our composition framework—with its techniques and methodology—to achieve this goal. Xcerpt modules can be called components in the more traditional sense of the word; they have clearly defined interfaces, ensure separation of concern and can be composed. In particular, a module programmer does not have to know about ISC, or ever hear the term 'fragment,' but can instead intuitively understand how to make use of this additional and convenient functionality by relying on more familiar notions such as 'modules,' and their 'input' and 'output.'

If the reader can be convinced that modules, as defined in this chapter, are useful tools for Web query developers—whether for separation of concern, reuse, or both—then it is interesting to consider the relatively small effort that went into enabling them. We deployed our presented composition framework, specified a component model and implemented the required composition operators. Being able to acquire an arguably important language feature by investing comparatively little is a powerful notion.

There are several improvements that could be done to Modular Xcerpt. The seemingly most interesting enhancement would be to be able to statically check the usage of modules. That is, to be able to inform query programmers of incorrect module usage. We will discuss this further in Section 8.2.

5.A Appendices

The first part of the appendix contains Xcerpt modules used in examples in this chapter. The second part contains the specification of the composition operators for the U-ISC–based composition system that realizes the Xcerpt modules.

Xcerpt modules

The following contains Xcerpt modules used in examples in this chapter. The location of the module as used in the examples is also indicated.

Amazon module: `file:import/Amazon.rxcerpt`

```
MODULE Amazon

CONSTRUCT
    public cd [
        artist [ var Artist ],
        title [ var Title ],
        coverlink [ var Coverlink ],
        songs [ all song [ var Songtitle ] ]
    ]
FROM
    public html [
        head [[ ]],
        body [[
            var Artist, br [],
            var Title, br [],
            img {
                attributes {
                    src { var Coverlink }
                }
            },
            table [[
                tr [ th [[ ]] ],
                tr[
                    td [ var Songtitle ],
                    td [[ ]]
                ]
            ]]
        ]]
    ]
END
```

MobileLayout: `file:html/layout/MobileLayout.rxcerpt`

```
MODULE MobileLayout

CONSTRUCT
    public table [
        all tr [ td [ attributes { class { "artist" } }, var Artist ],
                td [ var Title ] ] ]
FROM
    public input [[
            cd [[
                artist [ var Artist ],
                title [ var Title ]
```

```
   ]] ]]
END
```

DesktopLayout: `file:html/layout/DesktopLayout.rxcerpt`

```
MODULE DesktopLayout

CONSTRUCT
   public table [
          attributes { class { "maintable" } },
          all var Data
   ]
FROM
   reordereddata [[ dataset [[ var Data ]] ]]
END

CONSTRUCT
   reordereddata [
      all dataset [
         tr [
            attributes {
               class { "heading" }
            },
            td [
               attributes {
                  colspan { "2" }
               },
               var Artist, ":", var Title
            ]
         ],
         tr [
            attributes {
               class {"value"}
            },
            td [
               img[
                  attributes {
                     src { var Coverlink },
                     width { "250" },
                     height { "250" }
                  }
               ]
            ],
            td [
               table [
                  attributes {
                     class { "SongTitles" }
                  },
                  all tr [
                     td [ var Song ]
                  ]
               ]
            ]
         ]
      ]
   ]
FROM
   public input [[
      cd [[
         artist [ var Artist ],
```

```
            title [ var Title ],
            coverlink [ var Coverlink ],
            songs [[ song [ var Song ] ]]
        ]]
    ]]
END
```

StylesheetCreator: `file:html/style/Stylesheet.rxcerpt`

```
MODULE StyleSheet

CONSTRUCT
    public link [
        attributes {
            rel { "Stylesheet" },
            type { "text/css" },
            href { var Style }
    } ]
FROM
    public style [ var Style ]
END
```

RssLayout: `file:rss/RssLayout.rxcerpt`

```
MODULE RssLayout

CONSTRUCT
    public item [
            title [ var Artist, "-", var Title ],
            description [ "No description" ]
    ]
FROM
    public input [[
            cd [[
                artist [ var Artist ],
                title [ var Title ]
    ]] ]]
END
```

HtmlCreator: `file:html/HtmlCreator.rxcerpt`

```
MODULE HtmlCreator
IMPORT file:html/style/Stylesheet.rxcerpt AS StyleSheet

CONSTRUCT
    public html [
            attributes {
                xmlns { "http://www.w3.org/1999/xhtml" },
                lang { "en" }
            },
            head [ var Title, optional var Style ],
            var Body
        ]
FROM
    or {
        webpagedata [ var Title, var Body ],
```

```
        style [ var Style ]
    }
END

CONSTRUCT
    webpagedata [ var Title, var Body ]
FROM
    public creator [[
            var Title -> title [[ ]],
            var Body -> data [[ ]]
    ]]
END

CONSTRUCT
    style [ var Style ]
FROM
    IN StyleSheet ( var Style )
END

CONSTRUCT
    TO StyleSheet ( var Data )
FROM
    public creator [[ var Data -> style [[ ]] ]]
END
```

RssCreator: `file:rss/RssCreator.rxcerpt`

```
MODULE RssCreator

CONSTRUCT
    public rss [
        channel [ var Title, all var Item ]
    ]
FROM
    rssdata [ var Title, var Item ]
END

CONSTRUCT
    rssdata [ var Title, var Item ]
FROM
    public creator [[
        var Title -> title [[ ]],
        data [ var Item ]
    ]]
END
```

Modular Xcerpt composition operators

```
package org.reuseware.air.language.xcerpt.ops;

import java.util.Hashtable;

import org.reuseware.air.algebra.fragment.FragmentSystem;
import org.reuseware.air.coconut.IComplexOperator;
import org.reuseware.air.coconut.ReusewairComposer;
import org.reuseware.air.language.xcerpt.IConstructTerm;
import org.reuseware.air.language.xcerpt.IInInterface;
import org.reuseware.air.language.xcerpt.IModuleDefinition;
```

```
11 import org.reuseware.air.language.xcerpt.IName;
12 import org.reuseware.air.language.xcerpt.IOutInterface;
13 import org.reuseware.air.language.xcerpt.IQueryTerm;
14 import org.reuseware.air.language.xcerpt.IResource;
15 import org.reuseware.air.language.xcerpt.IXcerptStatement;
16 import org.reuseware.air.language.xcerpt.algebra.XcerptVisitor;
17 import org.reuseware.air.language.xcerpt.impl.IConstructTermImpl;
18 import org.reuseware.air.language.xcerpt.impl.IQueryTermImpl;
19 import org.reuseware.air.language.xcerpt.impl.IXcerptStatementImpl;
20
21 import de.tudresden.reuseware.language.rxcerpt.RxcerptPackage;
22 import de.tudresden.reuseware.language.xcerpt.XcerptPackage;
23
24 public class Composers implements IComplexOperator {
25
26   // communication information between operators
27   static Hashtable<String,String> names = new Hashtable<String,String>();
28
29   /**
30    * Required by IComplexOperator
31    */
32   public void initialize() {
33
34     FragmentSystem.getInstance().setGrammar("rxcerpt");
35     names.clear();
36   }
37
38   /**
39    * Constructor
40    */
41   public Composers() {
42     FragmentSystem.getInstance().setGrammar("rxcerpt");
43   }
44
45   /**
46    * ImportStatement Composer
47    */
48   @ReusewairComposer("ImportStatement")
49   public static IXcerptStatement importStatement(IModuleDefinition module, final
         IName name) {
50
51     /**
52      * 1) Extract module statements
53      *
54      */
55     IXcerptStatement stmt = new IXcerptStatementImpl();
56     module.accept(new XcerptVisitor(stmt) {
57
58       public boolean visit(IXcerptStatement node) {
59         getParamFragment().extend(node);
60         return true;
61       }
62
63       /**
64        * For composer communication
65        */
66       public boolean visit(IName node) {
67
68         if (node.inContextOf(RxcerptPackage.Literals.MODULE_DEFINITION) &&
69             !node.inContextOf(XcerptPackage.Literals.XCERPT_STATEMENT)) {
70           /**
71            * Save the connection between the prefix name and
```

```
72            * the name as defined by the module
73            */
74           if (!names.containsKey(name.toString()))
75             names.put(name.toString(), node.toString());
76         }
77         return true;
78       }
79     });
80
81     /**
82      * 2) Transform module statements (encapsulate)
83      */
84     stmt.accept(new XcerptVisitor() {
85
86       /*
87        * Construct Terms
88        */
89       public boolean visit(IConstructTerm node) {
90
91         //
92         if (node.inContextOf(RxcerptPackage.Literals.IN_MODULE) ||
93             node.inContextOf(RxcerptPackage.Literals.TO_MODULE))
94           return true;
95
96         boolean isTopLevel =
97           node.isContainedIn(XcerptPackage.Literals.CONSTRUCT_QUERY_RULE) ||
98           node.isContainedIn(XcerptPackage.Literals.CONSTRUCT_FACT);
99
100        if (isTopLevel) {
101          final String id = "\"" + names.get(name.toString()) + "\"";
102
103          // use internal fragment
104          final IConstructTerm localConstructTerm =
105            IConstructTermImpl.load("store [ id [ <<id>> ], section [ <<vis>> ],
                 <<cTerm>> ]");
106
107          // check if we are given information from the Import operator
108          if (names.containsKey(name.toString())) {
109            localConstructTerm.bind("id", IConstructTermImpl.load(id));
110            localConstructTerm.bind("cTerm", node);
111            // set visibility
112            if (node.inContextOf(RxcerptPackage.Literals.OUT_INTERFACE)) {
113              localConstructTerm.bind("vis", IConstructTermImpl.load("visout[]"));
114            } else {
115              localConstructTerm.bind("vis", IConstructTermImpl.load("vispriv[]"));
116            }
117            // replace
118            node.bind(localConstructTerm);
119          }
120        }
121        return true;
122      }
123
124      /*
125       * Query Terms
126       */
127      public boolean visit(IQueryTerm node) {
128
129        // handle different special cases
130        if (node.isType(XcerptPackage.Literals.IN_RESOURCE))
131          return true;
132
```

```
133       if (node.inContextOf(RxcerptPackage.Literals.IN_MODULE))
134         return true;
135
136       boolean containedInDisj =
              node.isContainedIn(XcerptPackage.Literals.DISJUNCTION_QT);
137       boolean containedInConj =
              node.isContainedIn(XcerptPackage.Literals.CONJUNCTION_QT);
138       boolean isTopLevel =
139         node.isContainedIn(XcerptPackage.Literals.CONSTRUCT_QUERY_RULE) ||
140         node.isContainedIn(XcerptPackage.Literals.GOAL_QUERY_RULE);
141
142       boolean isConjunction = node.isType(XcerptPackage.Literals.CONJUNCTION_QT);
143       boolean isDisjunction = node.isType(XcerptPackage.Literals.DISJUNCTION_QT);
144
145       if ((containedInDisj && !isConjunction) ||
146           (containedInConj && !isDisjunction) ||
147           (isTopLevel && !(isConjunction || isDisjunction)) ||
148           (node.isContainedIn(RxcerptPackage.Literals.IN_INTERFACE)))
149       {
150
151         // use internal fragment
152         final IQueryTerm localQueryTerm =
153           IQueryTermImpl.load("store [ id [ <<id>> ], section [ <<vis>> ],
                  <<qTerm>> ]");
154         final String id = "\"" + names.get(name.toString()) + "\"";
155
156         // check if we are given information from the Import operator
157         if (names.containsKey(name.toString())) {
158           localQueryTerm.bind("id", IQueryTermImpl.load(id));
159           localQueryTerm.bind("qTerm", node);
160           // set visibility
161           if (node.inContextOf(RxcerptPackage.Literals.IN_INTERFACE)) {
162             localQueryTerm.bind("vis", IQueryTermImpl.load("visin[]"));
163           } else {
164             localQueryTerm.bind("vis", IQueryTermImpl.load("vispriv[]"));
165           }
166           // replace
167           node.bind(localQueryTerm);
168         }
169       }
170       return true;
171     }
172   });
173
174   // replace passive syntax
175   stmt.accept(new XcerptVisitor() {
176
177     public boolean visit(IOutInterface node) {
178       IConstructTerm cTerm = new IConstructTermImpl();
179       node.accept(new XcerptVisitor(cTerm) {
180
181         public boolean visit(IConstructTerm node) {
182           if (node.inContextOf(RxcerptPackage.Literals.OUT_INTERFACE))
183             getParamFragment().bind(node);
184           return true;
185         }
186       });
187
188       if (cTerm.isLoaded())
189         node.bind(cTerm);
190
191       return true;
```

```
192     }
193
194     public boolean visit(IInInterface node) {
195       IQueryTerm qTerm = new IQueryTermImpl();
196       node.accept(new XcerptVisitor(qTerm) {
197
198         public boolean visit(IQueryTerm node) {
199           if (node.inContextOf(RxcerptPackage.Literals.IN_INTERFACE))
200             getParamFragment().bind(node);
201           return true;
202         }
203       });
204
205       if (qTerm.isLoaded())
206         node.bind(qTerm);
207
208       return true;
209     }
210   });
211   // default
212   return stmt;
213 }
214
215 /**
216  * ImportStatement Composer
217  */
218 @ReusewairComposer("ImportWith")
219 public static IXcerptStatement importWith(IModuleDefinition module, final
       IName name,
220     IName slot, IResource resource)
221 {
222   // do the standard import
223   IXcerptStatement stmt = importStatement(module, name);
224   // bind slot
225   stmt.bind(slot.toString(), resource);
226
227   return stmt;
228 }
229
230 /**
231  * InModule Composer
232  */
233 @ReusewairComposer("InModule")
234 public static IQueryTerm inModule(IName name, IQueryTerm queryTerm) {
235
236   // use internal fragment
237   final IQueryTerm localQueryTerm =
238     IQueryTermImpl.
239         load("store [ id [ <<id>> ], section [ <<vis>> ], <<qTerm>> ]");
240   final String id = "\"" + names.get(name.toString()) + "\"";
241
242   // check if we are given information from the Import operator
243   if (names.containsKey(name.toString())) {
244     localQueryTerm.bind("id", IQueryTermImpl.load(id));
245     localQueryTerm.bind("vis", IQueryTermImpl.load("visout[]"));
246     localQueryTerm.bind("qTerm", queryTerm);
247   } else {
248     System.err.println("The module name '" + name.toString() + "' has not been
         declared");
249   }
250   return localQueryTerm;
251 }
```

```
252
253  /**
254   * ToModule Composer
255   */
256  @ReusewairComposer("ToModule")
257  public static IConstructTerm toModule(IName name, IConstructTerm
         constructTerm) {
258
259    // use internal fragment
260    final IConstructTerm localConstructTerm =
261      IConstructTermImpl.
262          load("store [ id [ <<id>> ], section [ <<vis>> ], <<cTerm>> ]");
263    final String id = "\"" + names.get(name.toString()) + "\"";
264
265    // check if we are given information from the Import operator
266    if (names.containsKey(name.toString())) {
267      localConstructTerm.bind("id", IConstructTermImpl.load(id));
268      localConstructTerm.bind("vis", IConstructTermImpl.load("visin[]"));
269      localConstructTerm.bind("cTerm", constructTerm);
270    } else {
271      System.err.println("The module name '" + name.toString() + "' has not been
             declared");
272    }
273    return localConstructTerm;
274  }
275 }
```

In order for ontologies to have the maximum impact, they need to be widely shared. In order to minimize the intellectual effort involved in developing an ontology they need to be reused. In the best of all possible worlds they need to be composed.

OWL Web Ontology Language Guide (2004)

6

Ontology Components: Role Models for Ontologies

[*This chapter is closely based on [41] and [74], with slight refinements and a concrete specification of a composition system for role models.*]

The term 'ontology' originates from philosophy and describes the dealings with the nature of being. In the context of software/system engineering, the term is used to describe formalized vocabularies (terminologies), or to represent shared domain descriptions [35]. Ontologies are already deployed in the life sciences and the Semantic Web, but are expected to be deployed in many other areas in the near future—for example, in software development. As the use of ontologies becomes commonplace, they will be constructed more frequently and also become more complex. To cope with this issue, modularization paradigms and reuse techniques must be defined for ontologies and supported by ontology languages. One issue that always must be tackled when dealing with modularization is the question of what constitutes a module; what are the appropriate units from which larger ontologies can be built? In this chapter we propose to use *role models* from conceptual modeling for this purpose, and show how they can be used to define ontological reuse units and enable modularization.

In conceptual modeling it has long been known that there is a fundamental distinction between different kinds of concepts: some stand on their own (e.g. *Person*), while others depend on the existence of some other concept (e.g. *Borrower*, who must be related to the borrowed item). Making this distinction explicit is favored in the role modeling community (see e.g. [83, 84] and references therein), with successful applications, for example, in object-oriented programming [44]. In role modeling, concepts that can stand on their own are called *natural types*, while dependent concepts are called *role types*. Even though considered an important and fundamental conceptual differentiation, current ontology languages, such as OWL DL [73], do not support it (nor does the OWL 2 working draft [26]). The Description Logics (DLs) community—providing much of the foundations for OWL DL—is however aware of the differentiation and refers to role types as *relationship-roles* [8].[1] The DL handbook even supports the

[1] The DL community uses the term 'role' for binary properties. To avoid confusion with conceptual roles

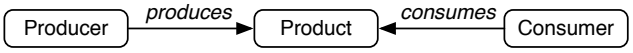


FIGURE 6.1: *A simple role model describing three related role types and their relationships.*

idea for the ontology development process by encouraging its readers to "distinguish independent concepts from relationship-roles." [8, p. 379]

Distinguishing different kinds of concepts is not only important for a better understanding of the modeled domain, but also for ontology reuse. This second application of role types has—to the best of our knowledge—never been investigated by the ontology community. Related role types and their relationships form abstraction units that can be studied and defined on their own. Such abstraction units are traditionally called *role models*. As role models often transcend domains, they can be reused in different ontologies.

Consider, for example, the simple role model in Figure 6.1. It describes the role types *Producer*, *Product*, and *Consumer*, and their relationships.

This role model could be integrated in ontologies covering topics as different as foods, wines, consumer electronics, or car dealerships. For example, a consumer electronics company modeling their infrastructure and their business processes can (re)use the generic description captured in the role model in Figure 6.1.

Role modeling can be seen as a design methodology, or process, allowing to break a larger description into smaller, reusable units—the role models. But to successfully deploy design processes for reuse, the underlying languages must support them. To quote Kiczales et. al. [51]: "Software design processes and programming languages exist in a mutually supporting relationship." Clearly 'programming languages' can here be substituted with 'ontology languages.' They go on to explain that the design processes allow to break a system down into smaller and smaller (reusable) units, and that "a design process and a programming language work well together when the programming language provides abstraction and composition mechanisms that cleanly support the kinds of units the design process breaks the system into." [51] It would hence be advantageous—from a reuse point of view—if ontology languages supported the definition of role models and provided constructs for composing role models into complete ontologies.

In this chapter we describe how ontology languages enabled with constructs for role modeling can provide for an important and little investigated reuse opportunity—in form of role models. We also discuss the consequences of supporting the underlying role modeling semantics. The contribution of this chapter consists of two parts:

1. *Role models as reuse units for ontology languages.* Regarding modularization of ontologies we make the following contributions:

   (a) We demonstrate the viability of role models as ontological reuse units.

   (b) We propose a formalization for what role modeling means for current ontology languages.

   (c) We explain the consequences of introducing the semantics of role modeling into ontology languages. That is, the possible reasoning effects the

(relationship-roles) we will instead use the term *dl-role* to refer to them.

ontology engineer has to be aware of.

(d) We describe how the role modeling semantics can be realized by reducing role modeled ontologies into ontologies expressed in standard ontology languages. This enables the reuse of reasoning engines.

2. *Composition framework validation.* This work on ontology role modeling has been carried out in the light of the composition technology outlined in Chapters 2–4. We show how role modeling for ontology languages can be realized by instantiating our composition framework. That is, we demonstrate how component-based development for ontologies can be provided by embedded invasive software composition (E-ISC, and the underlying technologies on which it builds; see Chapter 4).

This chapter is structured as follows. In Section 6.1 we first introduce role modeling as a modeling paradigm, and then briefly the ontology language OWL and its underlying logical formalism. In Section 6.2 we motivate the use of role models for ontology languages, both conceptually and via an example. In Section 6.3 we describe two different semantics for role modeling in ontology languages, and discuss their different characteristics. In Section 6.4 we develop a composition system for composing role models and role-based ontologies. In Section 6.5 we discuss related work, and finally in Section 6.6 we conclude the chapter.

6.1 Background

This section gives background knowledge for the rest of the chapter. First, we introduce role modeling as it is known from conceptual and software modeling. Then, we discuss Description Logics, the formal underpinning of many current ontology languages.

6.1.1 Role Modeling

In modeling, types (or concepts) abstract over sets of individuals.[2] Role modeling at its core argues for the existence of two inherently distinguishable abstractions: *natural types* and *role types*, a terminology first introduced by Sowa [81]. Intuitively, natural types describe the part of individuals that are essential to their identity while role types describe accidental or temporal relations to other individuals.

A common example is the natural type *Person* and its associated role type *Actor*. In this case, a person is said to *play the role* of an actor. Along with being an actor comes, for example, giving performances led by stage managers and attending rehearsals led by directors. Hence, in the role of being an actor (instance of concept *Actor*) one stands in certain relations to individuals of other role types, such as *StageManager* and *Director*.

Guarino defines natural types and role types using the notions of *founded types* and *semantically rigid types* [36]. A type is founded if all of its individuals have to be related to an individual of another type, where the relation is not part-of. For example, one could say that an actor necessarily has to be related to a director in order to be an actor. A type is semantically rigid if it contributes to the identity of its individuals. For instance, the name and date of birth of persons are part of their identity. This means

[2] In object-oriented modeling, the terms *class* and *object* are used instead.

#	Property	Comment
S1.	*A role comes with its own properties and behavior.*	Roles are types.
S2.	*Roles depend on relationships.*	A role is particularly meaningful in context of a relationship.
S3.	*An object may play different roles simultaneously.*	Since roles are types, this means multiple classification.
S4.	*An object may play the same role several times, simultaneously.*	Each occurrence of an object in a role is associated with a different state.
S5.	*An object may acquire and abandon roles dynamically.*	
S6.	*The sequence in which roles may be acquired and relinquished can be subject to restrictions.*	For example, a person can become a teaching assistant only after having been a student.
S7.	*Objects of unrelated types can play the same role.*	
S8.	*Roles can play roles.*	For example, a person can play the role of an employee, who in turn can play the role of a project leader.
S9.	*A role can be transferred from one object to another.*	For example, the role of president can be transferred from one person to the next.
S10.	*The state of an object can be role-specific.*	The state of an object may vary depending on the role in which it is being addressed.
S11.	*Features of an object can be role-specific.*	Attributes and behavior of an object may be overloaded on a role-basis.
S12.	*Roles restrict access.*	When addressed in a specific role, part of the object is invisible.
S13.	*Different roles may share structure and behavior.*	Role definitions can inherit from each other.
S14.	*An object and its roles share identity.*	An object and its roles are the same.
S15.	*An object and its roles have different identities.*	An object and its roles are not the same (contradictory to 14).

TABLE 6.1: *Steimann's 15 role modeling properties.*

Static	*Undecided*	**Dynamic**
S1, S2, S3, S4, S7, S8, S12, S13	S10, S11, S14, S15	S5, S6, S9

TABLE 6.2: *Static vs. dynamic role properties.*

that a human being cannot drop the type *Person*, but ceasing to be an instance of *Actor* is possible. Using these two notions, we can define natural types and role types:

- A *natural type* is non-founded and semantically rigid.
- A *role type* is founded and semantically non-rigid.

Although the notion of roles seems intuitively clear, different definitions exist in the literature. Steimann summarizes—repeated in Table 6.1—the fifteen most common characteristics the research community associates with roles in object-oriented and conceptual modeling [83].

One often recurring characteristic of roles is their *dynamism*, or connection to behavior (at least properties S5, S6, S9, possibly S10, S11). That is, that objects constantly change between the different roles they can play. This notion also comes through in the often used simile of modeled objects participating in a play, acting out their respective (dramatic) roles, and switching roles depending on the particular scene. Another set of commonly used properties focuses rather on what it means to play a role and if there are restrictions in doing so; but perhaps in particular which *relations* an object stands in with other objects when playing the particular role (at least properties S1, S2, S3, S4, S7, S8, S12, S13, possibly S14, S15). The separation of Steimann's role categorizations along this dimension—dynamic vs. static—is depicted in Table 6.2.

As an ontology describes a static view of the world, those referring to dynamic aspects observable in software systems are not our focus. Among the remaining, we consider the S1, S2, S3 and S14 to be the most fundamental. This because they capture what roles are (S1), and that role types are inherently connected to relationships, which is, as we shall see, important for reuse (S2; S3 implies the possibility of multiple relationships). Finally, we consider S14 to be innate to the underlying notion of role playing—an individual is the same when playing a role.

Roles alone are beneficial to separate inherent and accidental characteristics of individuals. But, encapsulating several related roles into a *role model* is where role modeling truly becomes useful. Results from the object-oriented software community [42, 76] show that role models are interesting units of abstraction, for mainly two reasons: First, role models focus on one specific concern of a domain, and hence, help in separating concerns. Second, role models are often reusable across domain boundaries because they can describe relations between individuals on a more general (non–domain-specific) level than natural types can. In this chapter we will focus on the second point, reuse of role models, and show how role models can serve as ontological reuse units.

An often discussed question is the relation of natural types and role types. Intuitively, natural types are related to role types via the "can play role" relationship, but the question is what semantics to associate with it. We will here use the $N \rhd R$ notation for the "can play role" relation, where N is a natural type and R a role type. Sowa originally proposed that role types are subtypes of natural types [82]. That is, we would

use the subsumption relationship ("IS-A") to explain $\rhd$, such that:

$$N \rhd R \;\equiv\; R \sqsubseteq N$$

where $\sqsubseteq$ represents the IS-A relationship. This interpretation is quite intuitive and works remarkably well, but not in all situations as we discuss in Section 6.3.3.

An alternative way of representing roles are separate individuals that are attached to individuals of natural types in some way. However, this is inconsistent with role feature S14, since a role-playing individual would be split into at least two separate individuals. For a detailed discussion, the reader is referred to [83].

6.1.2 Description Logics and OWL

Description Logics (DLs) are a family of knowledge representation formalisms, where most members are sub-languages of first-order logics. DLs are used to capture the important *concepts* and relations (*roles* in DL parlance) between individuals of the modeled domain. We will refer to (binary) relations in DL as *dl-roles* to distinguish them from the conceptual role types. Concepts and dl-roles can be described by complex *concept* (resp. *dl-role*) *descriptions* using the construction operators available in the particular DL.

The most widely used DL is the one underlying OWL DL [73]. To simplify the presentation, we do not cover datatypes here. An OWL DL *interpretation* is a tuple $I = (\Delta^I, \cdot^I)$ where the individual domain Δ^I is a nonempty set of individuals, and $\cdot^I$ is an individual interpretation function that maps (i) each individual name o to an element $o^I \in \Delta^I$, (ii) each concept name A to a subset $A^I \subseteq \Delta^I$, and (iii) each dl-role name RN to a binary relation $RN^I \subseteq \Delta^I \times \Delta^I$.

Valid OWL DL concept descriptions are defined by the DL syntax:

$$C \quad ::= \quad \top \mid \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \{o\} \mid \exists R.C \mid \forall R.C \mid \geqslant mR \mid \leqslant mR$$

The interpretation function $\cdot^I$ is extended to interpret $\top^I = \Delta^I$ and $\bot^I = \emptyset$. The concept $\top$ ($\bot$) is called *owl:Thing* (*owl:Nothing*) in OWL. The interpretation function can further be extended to give semantics to the remaining concept and dl-role descriptions (see [73] for details).

An OWL DL ontology consists of a set of *axioms*, including concept axioms, dl-role axioms and individual axioms.[3] A DL knowledge base consists of a TBox, an RBox and an ABox. A *TBox* is a finite set of concept inclusion axioms of the form $C \sqsubseteq D$, where C, D are concept descriptions. An interpretation I satisfies $C \sqsubseteq D$ if $C^I \subseteq D^I$. An *RBox* is a finite set of dl-role axioms, such as dl-role inclusion axioms ($R \sqsubseteq S$). The kinds of dl-role axioms that can appear in an RBox depend on the expressiveness of the ontology language. An interpretation I satisfies $R \sqsubseteq S$ if $R^I \subseteq S^I$. An *ABox* is a finite set of individual axioms of the form $a : C$, called *concept assertions*, or $\langle a, b\rangle : R$, called *dl-role assertions*. An interpretation I satisfies $a : C$ if $a^I \in C^I$, and it satisfies $\langle a, b\rangle : R$ if $\langle a^I, b^I\rangle \in R^I$.

Let C, D be concept descriptions, C is *satisfiable* wrt. a TBox T iff there exist an interpretation I of T such that $C^I \neq \emptyset$; C subsumes D wrt. T iff for every interpretation I of T we have $C^I \subseteq D^I$. A knowledge base Σ is *consistent* (*inconsistent*) iff there exists (does not exist) an interpretation I that satisfies all axioms in Σ.

[3] Individual axioms are called *facts* in OWL.

Human-readable syntax – Manchester OWL Syntax OWL has several syntaxes, but OWL ontologies are most commonly represented by XML serializations. Such serializations are machine readable, which is good for tooling and interoperability, but less appealing to end-users and ontology designers. Many end-users prefer to use the Manchester OWL syntax [46], which is more user friendly for non-logicians, and also supported by ontology editors such as Protégé.[4] In short, the Manchester syntax "tries to minimize syntactic constructs that are difficult to enter or understand" [46, p. 3]. For example, the conjunction (disjunction) of concepts C and D, rather than using the mathematical symbol $\sqcap$ ($\sqcup$), can be written:

```
C and D   (C or D)
```

Other concept constructors have similar intuitive English words that can be used. Ontology axioms can also be represented. The axiom defining concept C as a subconcept of D ($C \sqsubseteq D$) can be written:

```
Class: C SubClassOf D
```

The more complex concept definition:

$$Student \sqsubseteq Person \sqcap (= 1 hasAge) \sqcap (= 1 hasGender) \sqcap \forall hasGender.\{male, female\}^5$$

stating that all students are persons, having only one age, a single gender which can only be `male` or `female`, can be written as:

```
Class: Student
       SubClassOf: Person
                   and hasAge exactly 1
                   and hasGender exactly 1
                   and hasGender only {male, female}
```

There are other Manchester OWL constructs not detailed here, but they are intuitive to understand when seen in an example. More detail on the Manchester OWL syntax can be found in [46].

Modularization in OWL OWL is only equipped with, at best, rudimentary modularity constructs and support for component-based ontology development. OWL natively provides some facilities for reusing ontologies and ontology parts. First, a feature inherited from RDF [55] (upon which OWL is layered) is *linking*—loosely referencing distributed Web content and other ontologies using URIs. Second, OWL provides an `owl:imports` construct which syntactically includes the complete referenced ontology into the importing ontology. The linking mechanism is convenient from a modeling perspective, but is semantically not well-defined—there is no guarantee that the referenced ontology or Web content exists. Furthermore, the component (usually an ontology class) is small and often hard to detach from the surrounding ontology in a semantically well-defined way. Usually a full ontology import is required since it is unclear which other classes the referenced class depends on. The `owl:imports` construct can only handle complete ontologies and does not allow for partial reuse. Overall, OWL seems to be inflexible in the kind of reuse provided, especially regarding the granularity of components.

[4] `http://protege.stanford.edu/`

[5] Here $=$ is obviously the combination of $\leq$ and $\geq$.

6.2 Role Modeling for Ontology Languages

The main purpose of this section is to give an intuitive understanding of why role models are reusable entities and how they can be beneficial to ontology engineers. But before looking at an example of an ontological role model in Section 6.2.1, it is worthwhile asking the question if role modeling should be adopted for ontology languages irrespective of any reuse opportunities and benefits.

The by far most common usage of ontologies today is capturing a shared understanding of information between people and software agents ($\sim$70%), and to enable reuse of domain knowledge ($\sim$56%) [23]. In both cases, ontological modeling in large parts constitutes modeling the *real world*; possibly arbitrary real world occurrences in the first case (information) and specific areas of the real world in the second (domains). Modeling the world is the process of capturing the essential concepts in a certain area of interest and describing how they relate to each other. As in this process the "truth lies in the world" [7], it seems reasonable that any ontology language should provide constructs that correspond to the observed real world phenomena. This is also supported by research on the design of adequate conceptual modeling languages [38]. Hence, it is a reasonable question to ask if this is the case for today's ontology languages, in particular, OWL.

Investigating the most natural ways of expressing and talking about the world goes all the way back to formalizations of natural language, but more recently in the development of database models in the 60's and 70's. One data model—perhaps not widely known—is Bachman's *role data model*. He had the following to say about it ([9] as cited in [84]):

> "The basic claim of the role model is that it more closely represents the real world than the network model or any other well known model. [...] It is a model where the person describing the data can say more about the data and thus provides a better understanding of that data [...]"

To have the ability to "say more about the data" seems to be valuable for today's ontology languages, since they are used to represent the real world. In conclusion: since today's role modeling ideas are based on Bachman's role data model, it seems worthwhile to attempt to join those ideas with today's ontology languages, in particular with OWL.

6.2.1 Ontology Modularization with Role Models

We now give an example of an ontological role model and discuss the consequences of a role modeling approach to ontology design. Then, in Section 6.3, we discuss the semantics of such ontologies. The examples are written in Manchester OWL syntax [46], which has been extended for the purpose of defining and composing role models; the keywords of the extended constructs are underlined.

Listing 6.1 shows an ontology that models a faculty, introducing main concepts such as *Professor*, *FacultyMember*, and *PhDStudent*. The faculty is managed by a board which is described in the role model in Listing 6.2. A board consists of board members that elect a chairman.[6] The chairman can appoint one of the members as secretary. The ontology in Listing 6.1 imports the board role model and can so use

[6] A 'chairman' is here a person designated to preside over a meeting.

```
Ontology: http://ex.org/Faculty
  ImportRoles: http://ex.org/Board
  Class: FacultyMember
    CanPlay: BoardMember'
  Class: Professor
    SubClassOf: FacultyMember
    CanPlay: ChairMan'
  Class: PhDStudent
    SubClassOf: FacultyMember
  Individual: smith
    Types: Professor, Chairman'
  Individual: mike
    Types: PhDStudent, BoardMember'
```

LISTING 6.1: *Role-based ontology.*

```
RoleModel: http://ex.org/Board
  Role: BoardMember'
  Role: Chairman'
    SubClassOf: BoardMember' and
      electedBy' some BoardMember'
  Role: Secretary'
    SubClassOf: BoardMember'
  ObjectProperty: electedBy'
    Domain: Chairman'
    Range: BoardMember'
  ObjectProperty: appointedBy'
    Domain: Secretary'
    Range: Chairman'
```

LISTING 6.2: *Role model.*

the concepts it defines. Concepts and properties defined in the role model are marked with ′ to distinguish them from the concepts introduced in the base ontology.

One might ask why the board is described in a role model. The reason is that boards have a recognizable structure with a typical set of relationships that hold between entities in that context, regardless of the particular underlying domain. It therefore makes sense to detach the description of the board from the faculty ontology.

The ontology in Listing 6.1 is made up of standard DL constructs, save the *ImportRoles* and *CanPlay* constructs. The meaning of the *ImportRoles* construct is the obvious, making the role model available to the ontology. The *CanPlay* constructs are crucial since they define the relations between the base ontology and the role model. We refer to such connecting statements as *bridge axioms*. The role model in Listing 6.2 makes use of two additional constructs, *RoleModel* and *Role* that have the obvious meaning (defining a role model and a role, respectively). The URL of a *RoleModel* can be used to import it using the *ImportRoles* construct.

6.2.2 Methodology

Role modeling provides a methodology for developing ontologies, a methodology that we claim encourages good design and supports reuse. The following are the intuitive development steps that we propose for constructing a role-based ontology:

1. *Define base concepts.* Define a base ontology that contains the main concepts of the modeled domain. These concepts correspond to natural types of the domain. That is, each concept in the base ontology should be semantically rigid and non-founded. In our example, an ontology modeler would start by defining basic concepts of a faculty, such as *Professor* and *PhDStudent*. Notice that, in a different universe of discourse, *Professor* may itself be a role type (for example, for an underlying natural type *Person*).

2. *Identify role models.* Identify accidental or temporal relationships that individuals, abstracted by the base concepts, may participate in. Then, identify the contexts that those relationships appear in and what concepts (role types) are involved. Such contexts should be described in separate role models to be integrated into the base ontology.

(a) If a role model for the desired relationships already exists, it can be reused.

(b) If no fitting role model exists, then define it. This involves capturing the important role types in the context and defining the relationships between them. There will exist relationships, since role types are by definition semantically non-rigid and founded. For instance, the board role model contains the role types *Chairman'* and *BoardMember'*, which are related by the *electedBy'* property.

To ensure reusability of role models they have to be self-contained. In particular, each property defined in a role model must have its domain and range restricted to a role type from the same role model. This guarantees that each individual that participates in such a property actually belongs to a role type of the role model.

3. *Define bridge axioms.* Describe how the identified role models should be integrated into the base ontology by defining appropriate bridge axioms. A bridge axiom can bind a role type to a natural type, assert that an individual belongs to a certain role type, or assert two individuals to be connected via a property that is part of a role model. For example, we connect *Professor* and *Chairman'* through a *CanPlay* axiom, and the individual *smith* is asserted to be a *Chairman'*.

6.2.3 Role Models vs. Base Ontologies

We argue that it is beneficial to separate role models from base ontologies during the ontology design process. In particular, following the above methodology brings the following advantages:

- Modularization during development:
  - Base ontology development can focus on the main domain concepts and their hierarchical relations.
  - Role models can be defined, and refined, without necessarily focusing on the domain concepts, because role models typically transcend domains.
  - A role model focuses on a single context and important relationships holding between entities in this context.
- Reuse of role models:
  - As role models concentrate on a single concern, reuse is more likely than with complete ontologies that intermingle different concerns.
  - Role models constitute ontological modules. A base ontology can use role type names and property names of a role model, but not redefine them. Hence, a role model provides an interface, via the names of its role types and properties.

The role-based ontology in Listing 6.3 demonstrates the reusability of the board role model from Listing 6.2. The role model is being deployed in a different setting, this time in an ontology modeling a company, instead of a university. Since the *concern* that is captured in the role model appears in both domains, it can be reused.

```
Ontology: http://ex.org/Company
  ImportRoles: http://ex.org/Board
  Class: President
    CanPlay: ChairMan'
  Class: VicePresident
    CanPlay: Secretary'
  Class: CompanyAdvisor
    CanPlay: BoardMember'
  Individual: donald
    Types: President, Chairman'
  Individual: jane
    Types: VicePresident, BoardMember'
```

LISTING 6.3: *Role-based ontology reusing the role model from Listing 6.2.*

6.3 Semantics of Ontological Role Modeling

In Section 6.3.1 we formalize ontological roles and role models. Then, in Sections 6.3.2 and 6.3.3, we propose two possible semantics for the role modeling constructs used in the preceding section. The two semantics cover different aspects of role modeling and are realized by mapping role-based ontologies to different DL constructs.

6.3.1 Formalization of Role-Based Ontologies

We formalize role-based ontologies in three parts, based on the methodology described in Section 6.2. A base ontology only contains natural types. Role models define role types and their relationships. We use bridge axioms to combine a base ontology with role models into a role-based ontology.

Definition 6.1. *(*Base ontology*) A* base ontology *is a finite set of axioms in some DL, capturing concepts that are assumed to correspond to natural types.*

A base ontology is assumed to capture concepts that provide semantic rigidity for individuals of the modeled domain. Naturally, any properties that inherently relate such concepts are also introduced, as are concrete individuals. We do not commit to a particular DL, since this definition is general enough to cover many DLs.

Definition 6.2. *(*Ontological role model*) An* ontological role model *is a TBox where each concept name is considered a role type. Each concept name must be "related" to another concept name in the role model, either via a dl-role, or by at least one axiom (e.g. a subsumption axiom). All dl-roles must be domain and range restricted to a type from the role model.*

The restriction of "related" concept names prevents role models from being divided into subparts with pairwise disjoint signatures.[7] If such a division is possible, the role model should be split into separate role models. Intuitively, this restriction ensures that a role model only describes one concern.

Definition 6.3. *(*Role-based ontology*) A* role-based ontology $\mathcal{O} = (\mathcal{N}, \mathcal{R}, \mathcal{B})$ *is a triple where* $\mathcal{N}$ *is a* base ontology*,* $\mathcal{R}$ *is a finite set of* role models *and* $\mathcal{B}$ *a finite*

[7] What "related" means is not formalized. We recognize this formalization as important future work, but here stay with the intuitive notion, as described.

set of bridge axioms*. The base ontology and the role models have pairwise disjoint signatures. The bridge axioms in $\mathcal{B}$ are of the form:*

1. $N \rhd R$ *(*terminological bridge axiom*), or*
2. $R(a)$ *or* $S(a,b)$ *(*assertional bridge axiom*)*

where N is an arbitrary concept description, and a,b are individuals, in $\mathcal{N}$, and where R is a concept name (role type), and S a dl-role, in one of the role models in $\mathcal{R}$.

The $\rhd$ symbol reads "can play" and specifies that instances of a natural type can play a role of a certain role type. Assertional bridge axioms define individuals to belong to certain role types, or to be related to other individuals via some dl-role.

To be able to reuse existing tools, most importantly, reasoners, we define the semantics of role-based ontologies via reduction to the underlying DL. Thus, the reduction algorithm unambiguously defines the semantics of role-based ontologies by referring to the already understood model-theoretic semantics of DLs.

6.3.2 Conjunctive Role Modeling Semantics

The goal is to define a semantics for role-based ontologies that cover desirable properties of role modeling as a discipline. As a minimal requirement, the semantics should cover the Steimann criteria S1, S2, S3, and S14 described in Section 6.1.1. More importantly, we need to account for the differentiation between natural and role types according to the distinction made by Guarino [36]. That is, natural types are semantically rigid, while role types are not, and do not provide identity for its instances. We will address this by acknowledging that individuals cannot only be instances of role types. This suggests that role types that are not explicitly related to some natural type should—by definition—be unsatisfiable (empty in all models of the ontology). In this case, unbound role types can conveniently be detected by deploying standard ontology reasoners. Relations between natural and role types should explicitly be modeled by ontology engineers using terminological bridge axioms (that is, using the *CanPlay* construct).

The semantics of a role-based ontology $\mathcal{O} = (\mathcal{N}, \mathcal{R}, \mathcal{B})$ is here given by a transformation to an ontology $\mathcal{O}'$ in the DL of $\mathcal{N}$ according to the following transformation:

$$\begin{array}{rcl} \mathcal{O}' & = & \mathcal{N} \cup \mathcal{R} \\ & \cup & \{R \sqsubseteq N | N \rhd R \in \mathcal{B}\} \\ & \cup & \mathcal{B} \setminus \{N \rhd R | N \rhd R \in \mathcal{B}\} \\ & \cup & \{R \sqsubseteq \bot | R \in \mathcal{R} \wedge \neg \exists N : N \rhd R \in \mathcal{B}\} \end{array}$$

As can be seen, the translation scheme consists of four steps:

1. *Integration.* The base ontology and the role models (with pairwise disjoint signatures) are combined.
2. *Terminological bridge axioms.* Here we use the semantics proposed by Sowa for realizing the $\rhd$ relationship [82]. That is, if instances of a natural type N can play a role of a role type R, we specify R to be subsumed by N.
3. *Assertional bridge axioms.* All other bridge axioms are incorporated into $\mathcal{O}'$. That is, all the assertional bridge axioms.

```
Ontology: http://ex.org/Faculty
  Class: FacultyMember
  Class: Professor
    SubClassOf: FacultyMember
  Class: PhDStudent
    SubClassOf: FacultyMember
  Class: BoardMember'
    SubClassOf: FacultyMember
  Class: Chairman'
    SubClassOf: BoardMember' and
      electedBy' some BoardMember'
      and Professor

  Class: Secretary'
    SubClassOf: BoardMember' and
                owl:Nothing
  ObjectProperty: electedBy'
    Domain: Chairman'
    Range: BoardMember'
  ObjectProperty: appointedBy'
    Domain: Secretary'
    Range: Chairman'
  Individual: smith
    Types: Professor, Chairman'
  Individual: mike
    Types: PhDStudent, BoardMember'
```

LISTING 6.4: *Translation of a role-based ontology into the underlying ontology language.*

4. *Unbound role types*. Role types that are not related to any natural type through $\triangleright$ subsume $\bot$ (*owl:Nothing* in OWL), that is, they are unsatisfiable.

We will illustrate the transformation using the example from Listings 6.1 and 6.2. Applying the transformation yields the ontology in Listing 6.4. This ontology only uses standard ontology constructs. It must be highlighted that the ontology in Listing 6.4 only captures the *meaning* of the ontology units from Listings 6.1 and 6.2. The ontology engineer is never expected to continue working on the "compiled" ontology. The abstractions gained through explicit role modeling are lost in the compilation step, making the resulting ontology more difficult to maintain. At the same time, because the compilation result is expressed in a standard DL, existing ontology reasoners can handle it directly.

For instance, the role type *Secretary'* is not bound to any natural type, and hence, defined as a subtype of *owl:Nothing* ($\bot$). To understand the consequences of this translation, let us consider two scenarios:

1. Assume an additional assertion, stating that *mike* is an instance of *Secretary'*. As *Secretary'* is unbound, and thus, unsatisfiable, the resulting ontology would be inconsistent.

2. Assume another role type relies on instances of *Secretary'*. For instance, *Chairman'* could be a subclass of the concept `(assistedBy' some Secretary')`. As there can be no instances of *Secretary'*, *Chairman'* would also become unsatisfiable.

Obviously, our translation scheme for unbound role types helps in finding situations where role types are misused as natural types. The logical solution to repair our ontology for the two scenarios would be an additional bridge axiom *FacultyMember* $\triangleright$ *Secretary'*. In this case, *Secretary'* is no longer unsatisfiable and, as a consequence, the above illustrated inconsistencies do not occur.

Finally, let us come back to the four crucial properties of Steimann from Section 6.1.1. As we define roles as concepts that can be described by constructs of the underlying DL, we fulfill S1. The second property, S2, is also supported, since we require that the context of each role type is captured in its surrounding role model. The

second kind of bridge axiom in Definition 6.3 can be used arbitrarily often, hence, allowing one individual to be an instance of multiple role types (S3). The last property (S14) is supported as we do not represent roles as additional individuals but combine them with natural types using subsumption.

We refer to the above-described semantics as *conjunctive role modeling semantics* because a role type R played by different natural types $N_1, \ldots, N_n$ is interpreted as a subtype of the conjunction of the natural types, that is, $R \sqsubseteq N_1 \sqcap \ldots \sqcap N_n$. This follows immediately from defining $\rhd$ using standard subsumption. While simple, this semantics does not come without problems from a role modeling perspective. We investigate this further in the next section.

6.3.3 Disjunctive Role Modeling Semantics

Another arguably critical role modeling feature from Steimann's overview paper [83] is S7, repeated below:

S7 *Objects of unrelated types can play the same role.*

While Steimann claims that this property is "not acknowledged by all authors," it is a rather intuitive and useful modeling notion. Imagine, for example, that we replace the class *Professor*, from Listing 6.1, with the two classes *FullProfessor* and *AssistantProfessor*. Imagine, furthermore, that they are declared to be disjoint (natural, since you cannot be both). Suppose we want to express that both kinds of professors can be chairmen in a board. Not only is this a natural thing to express, but doing so would also enable us to discuss the properties of a chairman (defined by *Chairman'*), regardless of which kind of professor it is. To do this, we would issue the following bridge axioms:

$$FullProfessor \rhd Chairman'$$
$$AssistantProfessor \rhd Chairman'$$

While the above axioms are intuitive to understand and write, the conjunctive role modeling semantics, based on Sowa's original interpretation of $\rhd$, does not work as we perhaps would like. The reason is that the above gets interpreted as $Chairman' \sqsubseteq FullProfessor \sqcap AssistantProfessor$, which renders $Chairman'$ unsatisfiable. This is the case since the intersection of $FullProfessor$ and $AssistantProfessor$ is necessarily empty, since they are disjoint. In general, the conjunctive role modeling semantics can result in unexpected results when types that are not related via subsumption (e.g $FullProfessor$ and $AssistantProfessor$ in the above example) are related to the same role type via $\rhd$.

To be able to address S7, we provide an alternative semantics by letting role types be subsumed by the *union* of all natural types they are bound to. For the above example, this results in $Chairman' \sqsubseteq FullProfessor \sqcup AssistantProfessor$, which in this case does not make $Chairman'$ unsatisfiable. We call this the *disjunctive role modeling semantics* and its formal realization is as follows:

$$\begin{array}{rcl} \mathcal{O}' & = & \mathcal{N} \cup \mathcal{R} \\ & \cup & \{R \sqsubseteq N_1 \sqcup \ldots \sqcup N_n | \{N_1, \ldots, N_n\} = \{N | N \rhd R \in \mathcal{B}\}\} \\ & \cup & \mathcal{B} \setminus \{N \rhd R | N \rhd R \in \mathcal{B}\} \\ & \cup & \{R \sqsubseteq \bot | R \in \mathcal{R} \wedge \neg \exists N : N \rhd R \in \mathcal{B}\} \end{array}$$

The above definition only diverge from the corresponding definition for conjunctive semantics in the translation of terminological bridge axioms. Therefore, disjunctive

```
...
Class: FullProfessor
  SubClassOf: FacultyMember
Class: AssistantProfessor
  SubClassOf: FacultyMember
...
Class: Chairman'
  SubClassOf: BoardMember' and
    electedBy' some BoardMember' and
    (FullProfessor or AssistantProfessor)
...
```

LISTING 6.5: *Translation of a role-based ontology into the underlying ontology language using disjunctive role modeling semantics.*

semantics only differs from the conjunctive in cases where several natural types are bound to the same role type. In the case from the previous section with a single concept *Professor* bound to role type *Chairman'*, both semantics are equivalent. In contrast, the two semantics give different results if both *FullProfessor* and *AssistantProfessor* are bound to the role type *Chairman'*. While we obtain an inconsistency with the conjunctive semantics, disjunctive semantics allow for a consistent interpretation. The result is shown in Listing 6.5 (parts that are equal to Listing 6.4 are left out).

The disjunctive role modeling semantics satisfies S7, as well as the previously discussed role modeling requirements. Being able to connect unrelated, possibly disjoint, natural types to the same role type can be valuable from a modeling perspective. However, fulfilling S7 turns out to have a drawback: In contrast to standard DLs, the disjunctive semantics is non-monotonic. A logic is monotonic if adding a new axiom never falsifies assertions that were true before adding the axiom.

Theorem 6.1. *Ontological role modeling under disjunctive semantics is non-monotonic.*

The reason for this is that adding assertional bridge axioms can redefine previous knowledge. Consider the following role-based ontology $O = (\mathcal{N}, \mathcal{R}, \mathcal{B})$, with:

$$\begin{aligned}\mathcal{N} = \{&FullProfessor \sqcap AssistantProfessor \sqsubseteq \bot,\\ &FullProfessor(smith), AssistantProfessor(jones)\}\\ \mathcal{R} = \{&Chairman' = electedBy' \textbf{ some } BoardMember'\}\\ \mathcal{B} = \{&FullProfessor \rhd Chairman'\}\end{aligned}$$

Based on the disjunctive role modeling semantics we have $O \models \neg Chairman'(jones)$. But adding the bridge axiom $AssistantProfessor \rhd Chairman'$ to $\mathcal{B}$, this is not the case anymore, that is, $O \not\models \neg Chairman'(jones)$. As we have to retract knowledge when adding a bridge axiom, role modeling under the disjunctive semantics is non-monotonic.

6.4 Framework Evaluation: Composition System

In this section we specify a composition system for role modeling in Manchester OWL by instantiating the framework described in Chapters 2 and 4. We recall the devel-

opment process defined in Section 4.4.2 and go through the same development steps below.

D0 *Base grammar.* We first assume the existence of the grammar of the base language, in this case, Manchester OWL's grammar. We do not present the complete grammar here, but provide the overall structure that is needed to understand the component model specification in the next step. We only specify the abstract syntax.

⟨*Ontology*⟩ ::= ⟨*Identifier*⟩? ⟨*OntologyStatement*⟩*
⟨*OntologyStatement*⟩ ::= ⟨*ClassDescription*⟩ | ⟨*ObjectProperty*⟩ | ⟨*IndividualAssertion*⟩
⟨*ClassDescription*⟩ ::= ⟨*NamedType*⟩ ⟨*Description*⟩*
⟨*Description*⟩ ::= ⟨*SubClassOf*⟩ | ⟨*EquivalentTo*⟩ | ⟨*DisjointWith*⟩
⟨*SubClassOf*⟩ ::= ⟨*ClassExpression*⟩
⟨*EquivalentTo*⟩ ::= ⟨*ClassExpression*⟩
⟨*DisjointWith*⟩ ::= ⟨*ClassExpression*⟩
⟨*ClassExpression*⟩ ::= ⟨*AtomicExpression*⟩ | ⟨*Conjunction*⟩ | ⟨*Disjunction*⟩ | ⟨*Existential*⟩ | ⟨*Universal*⟩
⟨*AtomicExpression*⟩ ::= ⟨*NamedType*⟩
⟨*Conjunction*⟩ ::= ⟨*ClassExpression*⟩+
⟨*Disjunction*⟩ ::= ⟨*ClassExpression*⟩+
⟨*Existential*⟩ ::= ⟨*NamedProperty*⟩ ⟨*ClassExpression*⟩
⟨*Universal*⟩ ::= ⟨*NamedProperty*⟩ ⟨*ClassExpression*⟩
⟨*ObjectProperty*⟩ ::= ⟨*NamedProperty*⟩ ⟨*NamedType*⟩ ⟨*NamedType*⟩
⟨*IndividualAssertion*⟩ ::= ⟨*NamedIndiv*⟩ ⟨*IndividualTypes*⟩? ⟨*IndividualFacts*⟩?
⟨*IndividualTypes*⟩ ::= ⟨*NamedType*⟩+
⟨*IndividualFacts*⟩ ::= ⟨*IndividualFact*⟩+
⟨*IndividualFact*⟩ ::= ⟨*NamedProperty*⟩ ⟨*NamedIndiv*⟩
⟨*Identifier*⟩ ::= STRING
⟨*NamedType*⟩ ::= STRING
⟨*NamedProperty*⟩ ::= STRING
⟨*NamedIndiv*⟩ ::= STRING

The full Manchester OWL syntax defines some additional constructs, but these are not needed for our examples, or for understanding the following composition system definition. For more details on the Manchester OWL syntax, please consult [46].

D1–3 *Adapt base grammar to ISC, define domain-appropriate constructs and separate out active syntax.* The C_mSL^+ specification in Listing 6.6 extends the Manchester OWL base grammar for the purpose of working with role modeling and role models. The extension introduces intuitive constructs for defining and deploying role models, in line with the constructs used in Section 6.2 (see in particular Listings 6.1 and 6.2).

We only introduce one slot construct for `NamedType`. We will make use of this slot construct during the implementation of the composition operators. An example of this is shown in Listing 6.10.

```
extends file:owlm.gr @ o as file:rowlm.gr .

% slots
slotify o.NamedType .

% passive syntax
RoleModel           = modelID:o.NamedType, stmts:RoleStatement* .
RoleStatement       = RoleDefinition | RoleObjectProperty .
RoleDefinition      = roleID:o.NamedType, descriptions:o.Description* .
RoleObjectProperty  = roleprop:o.ObjectProperty.

% active syntax
ImportRoles         = rolemodel:RoleModel [ @ Location ] .
ImportRoles         <> o.OntologyStatement .
ImportRoles         -> @Composer .

CanPlay             = roleID:o.NamedType .
CanPlay             <> o.Description .
CanPlay             -> @Composer .

fragtypes { o.Ontology, o.OntologyStatement, o.ClassDescription,
            o.ClassExpression, o.ObjectProperty, o.Description, o.NamedType,
            RoleModel, RoleDefinition, CanPlay }
```

LISTING 6.6: C_mSL^+ *specification for extending Manchester OWL with a notion of role models.*

D4 *Generating component model.* Once the component model has been specified, it can be generated. This involves generating the core composition language—a Java API for Manchester OWL composition—enabling us, for example, to define ontology statements using the Java type `IOntologyStatement`. The kind of fragments that will be available to a developer in this manner are the ones enumerated using the `fragtypes` construct in Listing 6.6.

As an aside, it is perhaps interesting to note that an active syntax construct is listed in the `fragtypes` statement, namely, `CanPlay`. The reason for this is that the implementation of the composition operator for `CanPlay` needs to traverse the AST of the fragment being operated on. We discuss this issue further in Appendix 6.A, along with the definition of the composition operators (p. 194).

D5 *Defining composition operators for active syntax constructs.* The next step in the development process is to associate composition operators with the active syntax constructs introduced in the previous step. We notice that there are two such constructs defined by the nonterminals `ImportRoles` and `CanPlay`. Hence, we must define composition operators for these constructs. The operator implementations can be found in the appendix of this chapter. Here we look at one of them in detail, and leave the study of the other to the reader. The operator definition in Listing 6.10 corresponds to the `ImportRoles` construct. Notice, yet again, that the active syntax construct's implementation method signature directly corresponds to its definition in the component model.

The operator performs two main tasks:

1. *Role types as classes.* All role types in the imported role model are introduced as classes in the importing ontology. Each such role type is in

```
1 Ontology: file:Base.owlm
2   ImportRoles: file:Products.rowlm
3
4   Class: Computer
5   Class: Laptop
6     SubClassOf: Computer
```

LISTING 6.7: *Base ontology.*

```
1 RoleModel: file:Products.rowlm
2   Role: Product
3   Role: Warehouse
4   ObjectProperty: storedIn
5     Domain: Product
6     Range: Warehouse
```

LISTING 6.8: *Role model.*

```
1  Ontology: file:Base.owlm
2        Class: Product
3              SubClassOf: owl:Nothing
4        Class: Warehouse
5              SubClassOf: owl:Nothing
6        ObjectProperty: storedIn
7              Domain: Product
8              Range: Warehouse
9        Class: Computer
10       Class: Laptop
11             SubClassOf: Computer
```

LISTING 6.9: *Composed ontology.*

addition stated to subsume *owl:Nothing* (as the default state, until some natural type is bound to it). An internal fragment is defined to accomplish this (Line 8). It looks like this:

```
Class: «rolename» SubClassOf: owl:Nothing
```

When a role type is encountered in the imported role model, its name is bound to the slot `rolename` in the internal fragment (Line 15). In addition to creating a class for each role type, its existing definitions must be transferred (e.g. subclass-of declarations). This is done on Lines 19– 31. This whole task happens between Lines 6 and 36.

2. *Transfer object properties.* A role model does not only contain role type definitions, but also object properties relating the role types. These object properties must be made available in the resulting ontology and are hence transferred from the role model. This happens on Lines 38–42.

Notice that all the statements that are to be made available in the resulting ontology are "collected" into the `stmts` fragment (defined on Line 3). This collection of ontology statements is then returned as the result of the execution of the import operator (`ImportRoles`), and hence replace the call of the operator.

To clarify, let us look at a simple example. The base ontology in Listing 6.7 imports the role model in Listing 6.8

The result after composing the ontology in Listing 6.7—according to the composition operator implementation for `ImportRoles` in Listing 6.10—is the ontology in Listing 6.9 (in plain Manchester syntax).

The composed ontology is further transformed if `CanPlay` statements are specified. The operator implementation for `CanPlay` can be found in the appendix of this chapter.

```
public static IOntologyStatement importRoles(IRoleModel roleModel) {

    IOntologyStatement stmts = new IOntologyStatementImpl();
    roleModel.accept(new OwlmVisitor(stmts) {
        // introduce roles as classes
        public boolean visit(IRoleDefinition role) {
            // internal fragment
            IOntologyStatement cls = IOntologyStatementImpl.load(
                "Class: <<rolename>> SubClassOf: owl:Nothing");

            role.accept(new OwlmVisitor(cls) {
                // bind role type name
                public boolean visit(INamedType name) {
                    if (name.isContainedIn(RowlmPackage.Literals.ROLE_DEFINITION))
                        getParamFragment().bind("rolename", name);
                    return true;
                }
                // transfer class descriptions
                public boolean visit(final IDescription desc) {
                    if (desc.isContainedIn(RowlmPackage.Literals.ROLE_DEFINITION)) {
                        ((IOntologyStatement)getParamFragment()).accept(new
                             OwlmVisitor() {
                            public boolean visit(IDescription d) {
                                // only extend once!
                                if (d.isLast())
                                    d.prepend(desc);
                                return true;
                            }
                        });
                    }
                    return true;
                }
            });
            // extend collector fragment with statement
            getParamFragment().extend(cls);
            return true;
        }

        public boolean visit(IObjectProperty prop) {
            // extend collector fragment with object property statement
            getParamFragment().extend(prop);
            return true;
        }
    });
    // return all statements
    return stmts;
}
```

LISTING 6.10: *Composition operator specified for the active syntax construct* `ImportRoles` *that imports a role model into a base ontology.*

6.5 Related Work

Reuse is an important part in the ontology development process. A structured reuse opportunity is presented by *upper-level ontologies* (e.g. [67]). Upper-level ontologies typically describe generic concepts related to notions such as time, space, matter, and have a high reuse value since they span many different domains. Role models also span different domains, but their reuse usage is different from upper-level ontologies. In general, upper-level ontologies reside "above" the base ontology and reuse is accomplished by declaring a base concept B as a subclass of the upper-level concept U, that is, $B \sqsubseteq U$ in DL syntax. Role models on the other hand reside "below" the base ontology and the reuse relationship is inverted from the upper-level one. That is, reuse is in general accomplished using axioms such as $R \sqsubseteq B$, where R is a role type from a role model. As such, role models are *complementary* to upper-level ontologies. The main incentive for upper-level ontologies is to achieve base ontology integration and reuse is a condition for its success. For role models, reuse *is* the incentive.

Apart from the above comparison to the well-known idea of upper-level ontologies, our related work can in general be divided along two lines. First, the work can be aligned to other role modeling approaches. Then, it can also be compared to other approaches to ontology modularization. We make this separation below.

Role Modeling for Ontologies The OntoClean methodology proposed by Guarino shares a number of ideas with our approach [37]. The paper describes common misuses of the subsumption concept, for instance, to represent part/whole relations, instantiations, or meta-level relationships, and proposes to identify them using meta-properties for ontological classes. The two basic meta-properties of a class are *essence* and *rigidity*. Properties of a class belong to its essence, if they *must* hold for an instance (in contrast, for example, to properties of a role type, which *can* hold). Rigidity means that the properties of the class must hold for all instances. Our approach relates to this work as essential and rigid classes correspond to natural types, while non-essential and non-rigid classes are role types. Based on the meta-properties, Guarino proposes to impose constraints on subsumption relationships, for instance, forbidding a rigid concept to be subsumed by a non-rigid concept. This constraint is enforced in our approach by translating the terminological bridging axiom into $R \sqsubseteq C$, where a non-rigid concept R is always subsumed by a rigid concept C. Furthermore, OntoClean claims that each ontology has a backbone taxonomy with its rigid classes and their subsumption relationships. Such a backbone taxonomy roughly corresponds to our base ontology that exclusively consists of natural types.

The work in [85] proposes, similarly to us, to discriminate *role concepts* from *basic concepts* to overcome the gap between the recognition of different types of concepts and what is provided in standard ontology languages. The approach builds upon three notions: *role concepts*, equivalent to our role types, *potential players*, which roughly correspond to classes bound to a role type via a *CanPlay* relationship, and *role-holder*, that is, instances actually playing a role. The authors argue for two distinct type hierarchies and emphasize the relation of a role to its context. Furthermore, the paper describes compound roles that are built from primitive roles, realizing ideas that are similar to roles playing roles as in [83]. In contrast to our approach, the authors implement roles in their own ontology framework, including an ontology language and custom-built tools.[8] Instead, we propose to embed roles into existing languages using

[8] http://www.hozo.jp

syntactic extensions that can be translated into the underlying ontology language.

Ontology Modularization Modularizing ontologies and finding appropriate ontology reuse units are becoming important issues. Since one of our motivations is modularization of ontologies, we here mention some works addressing these issues, most having a strong formal foundation.

One work in this direction proposes a new import primitive: *semantic import* [70]. Semantic import differs from `owl:imports` (referred to as syntactic import) by allowing to import partial ontologies and by additionally enforcing the existence of any referred external ontologies and ontology elements by the notion of *ontology spaces*. The goal in this work is controlled partial reuse of ontologies; the reuse units are concepts, properties or individuals. The work in [25] defines a logical framework for modular integration of ontologies by allowing each ontology to define its *local* and *external* signature (that is, classes, properties etc.). The external signature is assumed to be defined in another ontology. Two distinct restrictions are defined on the usage of the external signatures. The first syntactically disallows certain axioms which are considered harmful, while the second restriction generalizes the first by taking semantical issues into consideration. The general goal, apart from a formal framework, is to allow safe merging of ontologies in a 'black-box' manner. The authors of [34] describe a technique for automatic partitioning of ontologies into reusable modules, under certain safeness conditions on the ontology. One interesting requirement put on such modules is that it should describe a well-defined subject matter, that is, be self-contained from a modeling perspective.

The first two mentioned approaches aim at reusing partial ontologies—entities defined within ontologies, in some sense *meaning*—rather than whole ontologies. Only the last approach seems to have an explicit desire for an ontology module to be a self-contained description of some subject matter (albeit not necessarily user-defined, but automatically constructed by the partition algorithm). In contrast, role models for ontologies are not units extracted from existing contexts and ontologies, but are defined by a modeler for the very purpose of being reused and deployed in varying contexts. Role models are also reuse units more coarse-grained than classes, resulting in ontologies constructed from fewer and larger parts. Most importantly, a role model has an intuitive reason for being a reusable part, which we argue is an essential feature for any modularization units of a modeling language.

6.6 Summary and Outlook

Ontologies are increasingly often applied in real-life scenarios, where they are used for modeling large knowledge domains. Perhaps the most prominent example to name here is the Gene Ontology [89] with its almost 25.000 terms (as of January 2008). To successfully develop and maintain such large ontologies, powerful modularization and reuse techniques are required.

In this chapter we have discussed the notion of roles as explicit modeling constructs. Roles have been discussed in the domain of conceptual modeling for some time, but never really utilized in ontology languages. We have shown how making roles explicit concepts—instead of encoding them implicitly in dl-roles—enables us to encapsulate role models and reuse them in different ontologies, even across domain boundaries. A role model in this context is an ontology consisting of role concepts and relationships between them. We have shown how to construct role-based ontologies from a base

ontology, a role model and a set of bridging axioms relating natural concepts from the base ontology to role concepts from the role model via a *CanPlay* relationship. The semantics of this new relationship has been defined by translating role-based ontologies to equivalent ontologies in a standard DL. We have discussed two such semantics: *conjunctive semantics* and *disjunctive semantics.* The former is simpler, but does not allow individuals of disjoint natural concepts to play the same role, which may be counterintuitive to ontology designers. The latter semantics allow such situations, but at the cost of a non-monotonic logic. It is at this stage not possible to recommend one of the semantics as canonical—they serve different purposes. We first need to study how role models are used and applied in practice to better understand which semantics is more intuitive to ontology engineers. Finally, we have instantiated our composition framework for providing an initial feel of using role models and role modeling for ontology design and construction.

In conclusion, we suggest to integrate explicit role concepts with ontology languages, such as OWL, as they offer a unique reuse and modularization opportunity that goes beyond currently available mechanisms for ontology reuse. Role models form *components*; their role types define a component interface enabling to check correct usage of the component—for example, every used role type must be assigned to some base concept. A notion of components that allows checking for correct usage is lacking in today's ontology languages. This gap can be closed by role modeling. Role modeling goes beyond upper-level ontology reuse, as it allows more specific ontologies to be reused and also allows reusing multiple role models within one ontology. Role models are, therefore, an important tool in every ontology designer's tool box. Steimann concludes his oft-cited article on roles with the following explanation and challenge [83]:

> "The challenge of defining a suitable role concept is to integrate it into existing modelling frameworks causing as little redefinition as necessary, while capturing as much of its semantics as possible."
>
> *– Friedrich Steimann [83, p. 104]*

One can view the referred modeling framework as consisting of ontology languages, in particular OWL and its underlying DL formalism. We argue that our approach of having roles as first-class constructs, but without extending the underlying formalism, is an initial answer to Steimann's challenge for ontology languages.

Of course, more work is needed. It would be advantageous to perform a larger case study of applying role models to re-factoring, modularizing, and partially reusing a large ontology. Role modeling should be supported by ontology modeling tools that can transform the role-based ontologies before applying reasoners. Furthermore, it will be interesting to study other applications of role models in ontologies—for example, where ontologies are used for describing situations in an action calculus [62].

6.A Appendices

This appendix contains the specification of the composition operators for the U-ISC–based composition system that realizes role modeling for Manchester OWL.

Role modeling composition operators

Below, the composition operators that correspond to the two active syntax constructs `ImportRoles` and `CanPlay` can be found. The code makes use of the core composition language (Java API) generated from the component models specification in Listing 6.6 (these Java classes have the namespace `org.reuseware.air.language.owlm`).

Discussion – active syntax constructs in `fragtypes` statement As promised, we will here briefly discuss the reasoning for including `CanPlay` in the `fragtypes` construct in the component model specification (cf. Listing 6.6).

To best support end-users and ontology engineers, it is desirable to have an intuitive syntax for working with role models. When extending an underlying language, any additional syntactical constructs should blend well together with the syntax of the underlying language. The syntax that needs to be taken into consideration for this example is the syntax of Manchester OWL. The `CanPlay` construct—binding role types to natural types—was decided to be a kind of Manchester OWL *description*. Existing Manchester OWL descriptions include `SubClassOf` and `EquivalentWith`. This means that we can use `CanPlay` in the following manner:

```
Class: NaturalType CanPlay: RoleType                    (6.1)
```

That is, wherever we can have a `SubClassOf` construct, we can have a `CanPlay` construct.

In addition, we decided that `CanPlay` should be an active syntax construct. This means that when a `CanPlay` construct is encountered during composition, its corresponding implementation method will be invoked on its arguments. The arguments in this case is the role type name to which the class should be bound (in (6.1) `RoleType` should be bound to `NaturalType`). What we actually want to do—to properly realize the defined role-modeling semantics—is to transform the imported role model where `RoleType` is defined (to define `RoleType` as subclass of `NaturalType`). And now comes the crux. In the implementation of `CanPlay`, the only real visible scope of the overall composing ontology (remember, composition is in progress) is the role type name (provided as an argument to the implementing method). But this operator needs to transform the composing ontology beyond this scope. How can this be dealt with? To get around this, we provide two additional methods that are defined, not on the normal fragment types (e.g. `INamedType`), but on the composition system specific algebra type (`<Grammar>Algebra`, cf. Section 3.3.2). In the case of Manchester OWL, the grammar name is assumed to be `Owlm`, so the Manchester OWL specific algebra type is named `OwlmAlgebra`. The two discussed methods defined on this type are:

```
OwlmAlgebra.getContainer()
OwlmAlgebra.getRoot()                                   (6.2)
```

However, the normal fragment types are subclasses of their corresponding algebra type, so the `getContainer()` method from (6.2) can be used as follows:

```
INamedType nt = ...
ICanPlay play = (ICanPlay)((OwlmAlgebra)nt).getContainer()   (6.3)
```

This gives the possibility to step outside of the current fragment scope. In (6.3) we went from a named type (here the name of a role type in a `CanPlay` construct) to the

surrounding `CanPlay` construct. The reason we want to do this is to find out how to transform the composing ontology (in this case we eventually want to know the natural type that the role type is specified to be bound to, cf. the composition operator implementation below). However, it order to be able to do this, the surrounding construct type we are inspecting must be supported by the underlying component model. Hence, in our case, we were forced to specify `CanPlay` in the `fragtypes` construct in the component model specification. If this is done, we can do something like (6.3). The `getRoot()` method inspects the top-most contextual construct – essentially the root node of the current abstract syntax tree.

Below follows the actual composition operator specifications.

```
1  package org.reuseware.air.language.owlm.ops;
2
3  import org.reuseware.air.algebra.fragment.FragmentSystem;
4  import org.reuseware.air.coconut.IComplexOperator;
5  import org.reuseware.air.coconut.ReusewairComposer;
6  import org.reuseware.air.language.owlm.IClassDescription;
7  import org.reuseware.air.language.owlm.IClassExpression;
8  import org.reuseware.air.language.owlm.IDescription;
9  import org.reuseware.air.language.owlm.INamedType;
10 import org.reuseware.air.language.owlm.IObjectProperty;
11 import org.reuseware.air.language.owlm.IOntologyStatement;
12 import org.reuseware.air.language.owlm.IRoleDefinition;
13 import org.reuseware.air.language.owlm.IRoleModel;
14 import org.reuseware.air.language.owlm.algebra.IOwlmFragment;
15 import org.reuseware.air.language.owlm.algebra.OwlmAlgebra;
16 import org.reuseware.air.language.owlm.algebra.OwlmVisitor;
17 import org.reuseware.air.language.owlm.impl.IClassExpressionImpl;
18 import org.reuseware.air.language.owlm.impl.IDescriptionImpl;
19 import org.reuseware.air.language.owlm.impl.INamedTypeImpl;
20 import org.reuseware.air.language.owlm.impl.IOntologyStatementImpl;
21
22 import de.tudresden.reuseware.language.owlm.OwlmPackage;
23 import de.tudresden.reuseware.language.rowlm.RowlmPackage;
24
25 public class Composers implements IComplexOperator {
26
27   /**
28    * Required by IComplexOperator
29    *
30    */
31   public void initialize() {
32     FragmentSystem.getInstance().setGrammar("rowlm");
33   }
34
35   /**
36    * Constructor
37    *
38    */
39   public Composers() {
40     FragmentSystem.getInstance().setGrammar("rowlm");
41   }
42
43   /**
44    * Methods corresponding to composers should be annotated with
45    * the following annotation: @ReusewairComposer, e.g.:
46    *
47    * @ReusewairComposer("[Name of composer construct]")
48    *
49    */
```

```
50
51  /**
52   * ImportStmt Composer
53   *
54   */
55  @ReusewairComposer("ImportRoles")
56  public static IOntologyStatement importRoles(IRoleModel roleModel) {
57
58    IOntologyStatement stmts = new IOntologyStatementImpl();
59    roleModel.accept(new OwlmVisitor(stmts) {
60      // introduce roles as classes
61      public boolean visit(IRoleDefinition role) {
62        // internal fragment
63        IOntologyStatement cls = IOntologyStatementImpl.load(
64          "Class: <<rolename>> SubClassOf: owl:Nothing");
65
66        role.accept(new OwlmVisitor(cls) {
67          // bind role type name
68          public boolean visit(INamedType name) {
69            if (name.isContainedIn(RowlmPackage.Literals.ROLE_DEFINITION))
70              getParamFragment().bind("rolename", name);
71            return true;
72          }
73          // transfer class descriptions
74          public boolean visit(final IDescription desc) {
75            if (desc.isContainedIn(RowlmPackage.Literals.ROLE_DEFINITION)) {
76              ((IOntologyStatement)getParamFragment()).accept(new OwlmVisitor() {
77                public boolean visit(IDescription d) {
78                  // only extend once!
79                  if (d.isLast())
80                    d.prepend(desc);
81                  return true;
82                }
83              });
84            }
85            return true;
86          }
87        });
88        // extend collector fragment with statement
89        getParamFragment().extend(cls);
90        return true;
91      }
92
93      public boolean visit(IObjectProperty prop) {
94        // extend collector fragment with object property statement
95        getParamFragment().extend(prop);
96        return true;
97      }
98    });
99    // return all statements
100   return stmts;
101 }
102
103 /**
104  * Plays Composer
105  *
106  */
107 @ReusewairComposer("CanPlay")
108 public static IDescription canPlay(final INamedType namedType) {
109
110   // get the contexts
111   ICanPlay playContainer =
```

```
112       (ICanPlay)((OwlmAlgebra)namedType).getContainer();
113     IClassDescription clsContainer =
114       (IClassDescription)((OwlmAlgebra)playContainer).getContainer();
115
116     // natural type
117     INamedType naturalType = new INamedTypeImpl();
118     clsContainer.accept(new OwlmVisitor(naturalType) {
119
120       public boolean visit(INamedType clsName) {
121         if (clsName.isContainedIn(OwlmPackage.Literals.CLASS_DESCRIPTION)) {
122           getParamFragment().bind(clsName);
123         }
124         return true;
125       }
126     });
127
128     // make the natural type a superclass of the appropriate role type
129     // get the main ontology
130     IOwlmFragment ontology = ((OwlmAlgebra)namedType).getRoot();
131
132     ontology.accept(new OwlmVisitor(naturalType) {
133
134       public boolean visit(final IClassDescription cls) {
135
136         cls.accept(new OwlmVisitor(getParamFragment()) {
137
138           public boolean visit(INamedType clsName) {
139             if (clsName.isContainedIn(OwlmPackage.Literals.CLASS_DESCRIPTION)) {
140               if (clsName.toString().equals(namedType.toString())) {
141                 // found the role type definition
142                 cls.accept(new OwlmVisitor(getParamFragment()) {
143                   public boolean visit(IClassExpression atom) {
144                     // check if there is already a disjunction
145                     if (atom.isContainedIn(OwlmPackage.Literals.DISJUNCTION)) {
146                       // only extend once!
147                       if (atom.toString().equals("owl:Nothing")) {
148                         atom.extend(IClassExpressionImpl.load(
149                           getParamFragment().toString()));
150                       }
151                     }
152                     // if not we can create a disjunction
153                     else if (atom.isType(OwlmPackage.Literals.ATOMIC_EXPRESSION)) {
154                       if (atom.toString().equals("owl:Nothing"))
155                         atom.bind(IClassExpressionImpl.load("( owl:Nothing or " +
156                             getParamFragment() + ")"));
157                     }
158                     return true;
159                   }
160                 });
161               }
162             }
163             return true;
164           }
165         });
166         return true;
167       }
168     });
169     // return something logically benign
170     return IDescriptionImpl.load("SubClassOf: Top");
171   }
172 }
```

Part IV

Summary

7 Related Work

This chapter discusses related work. The by far most related works are GBM as demonstrated in the Mjølner system [63], and ISC as realized by COMPOST [5]. Since we build upon and extend these works we do not further discuss them here. As explained, our approach is *grammar-driven* and relies on the existence of a base grammar. This is well in line with the declaration of the need of a more disciplined study of grammar-based software [54]:

– *Grammarware.* The authors of [54] call for a disciplined study of the role of grammars in engineering, or for the creation of an engineering discipline for "grammarware." Grammars are defined in an inclusive manner and cover context-free grammars, algebraic signatures and regular tree and graph grammars. The focus is on grammars as structural descriptions. Hence, the de facto interpretation of context-free grammars as sets of strings is not seen as valuable in this context since it does not emphasize the grammar's role to serve as a structural description. Interpreting a context-free grammar as the set of all valid derivation trees, however, does preserve the grammatical structure and is preferred. Our approach is labeled *meta-grammarware* since it "supports concrete grammar use cases by some means of metaprogramming." [54, p. 12]. Traditional meta-grammarware is a program generator that takes a possibly enriched grammar and produces an actual software component, e.g. a parser or a program transformer. Our deployment of the meta-grammarware concept is to take an enriched grammar (a base grammar and a component model specification) and produce a fragment-based composition system.

  While [54] enumerates different applications for grammar-based software, composition is not one of them. Nonetheless, our work clearly follows the spirit of grammarware.

Metaprogramming and (syntactic) program transformations are wide and active research areas, covering many different methods and approaches. Below we highlight works that we deem closely related and interesting to mention for comparison reasons.

We divide the presentation into three different categories: *Software transformation or generation techniques*, *Macro systems* and *Software composition approaches*.

Software transformation or generation techniques. There are several interesting works on extending *host languages* with DSLs (e.g. [12, 17, 45, 90]). The host languages are most often general-purpose languages, such as Java or Scala. There are several benefits to be gained from such an embedment approach. For example, reuse of existing functionality and constructs, existing language semantics can be exploited for the embedded language and existing infrastructure can be reused. The argument is that general-purpose languages already provide mechanisms for abstraction and modeling, while domain-specific languages offer dedicated constructs for expressing specific and domain-related notions. Thus, integrating an already rich host language with domain-specific constructs provides the required expressiveness and suitable syntax for the problem at hand. This stands in stark contrast to our original goal: extend non-embedded domain-specific languages, that lack more general constructs often found in general-purpose languages, with domain-appropriate (possibly non–domain-specific) abstractions.

– *Jakarta Tool Suite (JTS).* JTS is a tool suite for implementing DSLs [12]. It consists of two main parts: *Jak* and *Bali*. Jak is an extensible superset of Java that supports metaprogramming. Bali is a tool for composing grammars. A JTS *component* consists of a Bali grammar file defining the syntax of a language or extension, and a set of Jak files that define the semantics of the extension as syntactic transformations. The Bali tool can generate a set of Java classes corresponding to a specified grammar, that is, the AST structure of the grammar. These classes are then manually sub-classed to provide the reduction semantics of the Bali-specified extension. Using a predefined set of tree-traversal methods, Jak can then traverse ASTs and perform needed transformations. There are no restrictions on how the ASTs may be transformed.

  In comparison to complex composition operators that define the semantics of our language extensions, we do not implement the semantics via subclassing, and we have a component model that defines restrictions on how ASTs can be traversed. As regards Jak, adding support for metaprogramming with a new language is a manual process. Constructs are supported for particular languages (mainly Java) to build ASTs that are deployed during metaprogramming (e.g. `AST_Exp`, `AST_Stmt`, `AST_Class`). In contrast, we do not provide such tailored constructs to build ASTs (see concluding remarks in Section 3.6, p. 95).

– *ableJ.* The ableJ [90] framework can be used for extending a given host language with domain-specific constructs. The work aims at a tightly integrated and final composed language where problems can be formulated and solved with constructs most suitable for the job. Several language extensions can be added to the host language and the idea is that programmers are able to select the required extensions and have them automatically integrated. The host and extension languages are specified as attribute grammars in the Silver framework [96]. These specifications allow for semantical analysis across the different involved languages. [90] uses Java as the host language. Programs of a composed language are compiled to plain Java before byte compilation is performed by native Java compilers.

– METABORG. The work presented in [17] describes a method of integrating domain-specific languages in a host language. The approach is referred to as METABORG and integrates several other tools, for example, the Syntax Definition Formalism (SDF) [91] and Stratego/XT [92]. SDF is a formalism for specifying grammars, while Stratego/XT is a program transformation formalism. METABORG is based on the idea of extending a general-purpose host language with domain-specific constructs, and defining *assimilations* that describe how the extended constructs are mapped to the host language. The assimilation phase implements the introduced domain abstractions in terms of existing host language APIs, thus in a way bringing APIs to the language level. The assimilated code is guaranteed to syntactically fit the point of assimilation. It should be noted that the approach is not bound to a particular host language. The METABORG approach is powerful and useful for embedding DSLs in different host languages. METABORG itself is however a particular pattern of usage of other tools, in particular the mentioned SDF and Stratego/XT. In comparing to our work, Stratego/XT is of most interest and we discuss it below.

– *Stratego/XT.* Stratego/XT is a framework for implementing software transformation systems [92], used for example in [17] and [53]. Stratego consists of two parts: Stratego and XT. Stratego is the core of Stratego/XT and is a language for software transformation based on the paradigm of rewriting strategies. XT is a set of tools for generating parsers and pretty-printers etc. A Stratego specification is a set of term rewriting rules. Each rule, when matched on a certain input, produces the specified output. Stratego is very general and is a powerful language for transforming different kinds of formal texts into other formats. It is not limited to traditional executable programs. Stratego can for example be used to transform Java code into Java documentation (using Java's documentation annotations).

  The main divergence from our composition approach is the lack of a component model, or lack of transformational restrictions. We always employ a component model that restricts how entities may be transformed. From a transformational viewpoint, Stratego is a more general and powerful approach since it considers arbitrary transformations. Stratego/XT is not a component or composition approach. While the entities being transformed can be fragments, they are in no way components, and in particular lack any notion of interfaces.

– *Template engines.* A technique related to GBM is presented by template engines. One of the more prominent template engines today is STRINGTEMPLATE [71]. In STRINGTEMPLATE, a template specification can be seen as an *output grammar*. It is a logic-free specification of an output language. If the template is 'context-free,' it generates a context-free language. A template can contain *attributes* and *template applications*. Attributes can in this sense be seen as slots, which can be varied depending on the data being bound to them. The main application area for STRINGTEMPLATE is code-generation and Web page generation. For example, STRINGTEMPLATE allows for a clean separation between a Web page view, and the data populating the page. Template engines such as STRINGTEMPLATE are however more than just "documents with holes." STRINGTEMPLATE supports, for example, side effect-free expressions, template application to a list of data objects and nested template applications. There are some main differences between templates and GBM (or U-GBM). First, the ap-

plication areas are completely different and the approaches are developed for different purposes. Second, U-GBM is grammar-driven and founded on the existence of a base grammar that is adapted for composition. A template on the other hand is a manually specified output grammar for a specific language. Third, a template is usually an exemplar for an output, parameterized with particular data, and hence typically already contains many terminal symbols. A grammar used in GBM is a grammar in the more traditional sense, according to which some terminal strings are then later parsed. For a particular language (and its grammar), it seems reasonable that a template engine such as STRINGTEMPLATE *could* be used to achieve something similar to GBM, but somewhat awkwardly. Finally, template engines only consider explicit interfaces, while we move on from GBM to also consider implicit interfaces in the style of ISC.

Some of the above-mentioned approaches (in particular [17]) could in principle be used to achieve similar results as we do in E-ISC–based composition systems. However, they are general program transformation approaches, while we strive for a fragment-based composition approach (where fragments are components with interfaces).

Macro systems. Macro systems are closely related to the approach presented in this thesis. In particular, E-ISC–based language extensions can be seen as deploying a macro system. An overview of several macro systems and their properties is given in [15]. Macros allow to define new syntactical constructs that are transformed at precompile time. A macro application is replaced by its, possibly parameterized, definition. This process is called macro *expansion*. An example macro definition in Clojure is shown below:[1]

```
(defmacro times [x y] `(* ~x ~y))
```

Each location where the macro is called is replaced by the macro definition. For example, `(times 2 3)` will be replaced by the expression `(* 2 3)` before it is evaluated. One main difference between macros and functions is that function arguments are always evaluated before the function is called. This is however not the case with macros.[2] In the above example, the arguments `2` and `3` are not evaluated in the macro call. In a more complex macro definition, this can allow the macro to transform its arguments before returning an expression.

Macros can roughly be divided into *lexical* and *syntactical* [15]. Lexical macros do not consider any underlying language, in particular any grammar, during expansion. A problem in this case is that the macro expansion might result in a syntactically ill-defined program. A well-known example of such a macro system is the one available in the C programming language [50]. Syntactical marco systems on the other hand ensure that the result of a macro expansion is well-formed wrt. the underlying language, and from here on we only consider such macros. Macros define syntactic abstractions and operate on unevaluated code fragments, usually their AST representations. Hence, a macro accepts AST arguments and generates a new AST that replaces the macro invocation, at which point the evaluation/compilation of the program continues. One

[1] `http://clojure.org`

[2] One should avoid using macros where functions can be used.

approach to syntactical macros is presented in [15] and is closely related to our approach. While some macro systems only allow, and guarantee syntactical safety for, macros in certain positions in programs, the authors of [15] state that:

> "The ideal macro language would allow *all* nonterminals of the host language grammar to be extended with arbitrary new productions, defining new constructs that appear to the programmer as if they were part of the original language."
>
> *– Brabrand and Schwartzbach [15, p. 34]*

This goal is closely related to our work on E-ISC (cf. Chapter 4), both wrt. providing new constructs appearing to be part of the original language, and allowing to work on arbitrary positions in programs (hence, the reference to *all* nonterminals of the underlying language grammar). An example macro definition from [15] is the following:

```
syntax <stmt> si (<expr E>) <stmt S> ::= {
    if (<E>) <S>
}
```
(7.1)

where `stmt` and `expr` are nonterminals from the underlying language grammar. The **`syntax ... ::=`** `{ ... }` part is the concrete syntax for defining macros. The above macro defines an alternative if-statement, called `si`. The definition says that the `si`-construct takes two arguments of types `expr` and `stmt`, and returns a code fragment of type `stmt` (indicated by `<stmt>` between `syntax` and `si`). What is interesting in comparison to our approach is that this syntactical macro definition, with its types, directly corresponds to an active syntax construct (cf. Section 4.3, p. 111), defined as follows:

```
extends file:base.gr @ b as file:reuse.gr
Si ::= ex:b.expr, st:b.stmt .
Si <> b.stmt .
Si -> @Composer
```
(7.2)

In our case we only define the abstract syntax of the construct, but the idea is the same. The macro definitions in [15] must start with a keyword (in this case `si`), while we do not impose such restrictions (as long as it is possible to generate a parser for the extended language). In the same way as (7.1), (7.2) defines a construct (`Si`) taking two arguments of types `expr` and `stmt` and returning a `stmt` (determined by the "injection" construct). Macros always correspond to active syntax. Hence, in our approach, `Si` is a composer. The approaches differ much more in how the newly defined constructs are interpreted. In (7.1) a fragment template of the base language is specified with "holes" (the `if`-statement). Once the macro body template has been instantiated using the passed arguments as values, the system has to check that the resulting fragment can be parsed into a valid `stmt`. An equivalent macro interpretation would look something

like this in our approach:

```
public static IStmt interpretSi(IExpr ex, IStmt st){
    IStmt stmtFrag =
        new IStmtImpl.load("if (<<e:expr>>)<<s:stmt>>");
    stmtFrag.bind("e", ex);                                    (7.3)
    stmtFrag.bind("s", st);
    return stmtFrag;
}
```

As can be seen in (7.3), the macro expansion in this case consists of a set of explicit bindings. In (7.1) the macro template instantiation is not type safe, but the marco system then guarantees that the AST returned is of type `stmt` (checked during parsing of the marco body before returning it). In (7.3) the parsing is done as a first step, while the bindings of the parameters are guaranteed to be safe.

Clearly, the definitions in (7.2) and (7.3) are more verbose than the compact definition in (7.1). However, our approach has some benefits. One of them is that we have a "programmable" macro definition language. The bodies of macro definitions are in many approaches, including the one in [15], code templates (cf. (7.1)). At the other extreme, the macro system in Lisp allows arbitrary transformations [33]. It is clear that the expressiveness of the macro definition language is important. Our approach falls between these two extremes:

code templates < REUSEWARE/AIR < arbitrary transformations

The reason is that we always have an underlying component model. The component model specifies how the involved code fragments (their ASTs) may be transformed. By changing the component model, the expressiveness of the macro definition language is effectively changed. Hence, the macro definition language can be tailored depending on the needs of the system. In addition, we use Java as the language to write the macro definitions and as such do not give any termination guarantees as is done in [15] (the usefulness of which is questioned in [17, p. 380]).

The system in [15] introduces *metamorphisms* for the purpose of allowing arbitrarily long macro parameter lists. In our approach this is also possible, on the macro syntax definition side, but currently not on the macro implementation side (this is however only a technical limitation in the current realization, and not a conceptual one).

It should be added that we do not take static semantics of the macro expanded result into account, and neither does [15]. The approach in [15] needs to be implemented for each host language. In contrast, our approach is grammar-driven. With the help of the REUSEWARE framework, our approach can quickly integrate with different base languages. The work in [16] extends the work in [15] by allowing to easier work with different languages. Special parsing algorithms are developed for this purpose (called *specificity parsing*), while we rely on the existence of standard parsing libraries (such as ANTLR [72]).

One important property of macro systems is that they are "hygienic." A *hygienic* macro system guarantees not to cause collisions with existing symbol definitions (identifiers). If this is not guaranteed, unexpected and unwanted results can follow from macro expansions. This problem is usually solved by automatic renaming of symbol names used within macro definitions [12]. Our system does not provide such guarantees, since we do not currently see the need for them in our approach. The reason

why this is not directly needed is that we take a more metaprogramming approach where introduced identifiers (e.g. `stmtFrag` in (7.3)) reside in a different namespace from the actual fragments returned as results of macro expansions. In contrast, in e.g. [12, 15, 33] the macros are defined in a language closely related to the host language, and code in the macro definitions will end up at the locations of the macro invocations. That is, Lisp macros are defined in Lisp etc. We define, for example, Xcerpt macros in Java. Even if we composed Java fragments, the macro definitions would be specified on a meta-level. However, complications could arise from the use of *internal fragments* (cf. (7.3)). Hygienic guarantees must in this case be specified for each macro definition, or in a language-specific fragment composition library, since the general system does not know about identifiers in Java, or symbols in Lisp etc.

Other works on macros exists, but are less relatable to our work. These include [11, 31, 94]. Maya, presented in [11], describes a powerful method for extending Java syntax and reinterpreting its syntactic constructs. This system goes beyond traditional macro systems by also allowing to reinterpret base constructs.

While macro system have ideas in common with our approach, they are different in some fundamental ways. Our main aim is not merely to allow for the definition of new syntactic sugar. Instead, we aim to extend languages with constructs that can be used for component-based development, and specify their semantics using a controllable mechanism. This involves three issues not usually discussed in macro systems:

1. We allow for the definition of 'passive syntax,' which can be used to define components (whatever they may be) and their interfaces. Traditional macros only define 'active syntax.'

2. Extended language semantics is provided by the implementation of composition operators that correspond to active syntax. These composition operators are implemented in a controlled way, as dictated by a component model. This component model can quickly be changed, thus altering the implementation possibilities. Traditional macro systems have predefined constructs for specifying macro expansions, or it is left completely unrestricted.

3. For the realization of a component-oriented language extension, the overall extended language semantics can be realized by a set of *collaborating* composition operators. Macros are traditionally specified independently.

Nonetheless, our composition framework can be used to realize a kind of macros, although not as conveniently as some existing macro systems. The reason for this is that the approaches have different goals.

Software composition approaches. Next we study software composition approaches that relate to ours.

– *Aspect-oriented techniques.* Aspect-oriented programming (AOP) is a technique that increases modularization capabilities by allowing for the separation of different intermingled concerns [51]. A concern captures a particular part of the overall software realization. Such concerns can sometimes be sprinkled across the source code base, or intermingled with other concerns in the code. By separating out such concerns into cohesive units, they become understandable, maintainable and reusable. A concern formalized into an encapsulated unit is called an *aspect*. Common examples are aspects for logging and authentication. The

main demonstrator for AOP is the ASPECTJ system built for Java [2, 52, 61]. However, aspects are also investigated in other areas of software engineering, for example software modeling and product-line engineering (see e.g. [40]). However, we shall mainly relate to ASPECTJ in the following.

An aspect is integrated into the base code through a process called *weaving*. The implementation of the weaving is called *crosscutting*, since the weaving "cuts across" multiple modules in a systematic way. There are two types of crosscutting: *static* and *dynamic*. Dynamic crosscutting weaves new behavior into the execution of a program, while static crosscutting modifies the static structure of the system. The perhaps most important concept in AOP is the *join point model*. The join point model specifies which points in a system are identifiable for modification. The join point model for ASPECTJ is predefined and exposes the most important join points for Java. Locations in programs not supported by the join point model are not accessible. Join points also carry contexts with them. For example, a call to a join point in a method has access to the caller object, target object and arguments of the method. ASPECTJ is tightly connected to Java and contains many powerful constructs to support AOP on that platform.

Aßmann explains in [5] that ISC can be used to simulate AOP. ISC is a purely static technique that operates on the structure of source code. Hence, any crosscutting performed in a ISC-based system will be static. An example of a static crosscut is to weave an extra parameter into a method signature. However, many dynamic crosscuts can be simulated using static crosscutting. For example, we can alter the execution of a method by weaving in additional statements. Consider the following weaving:

```
1 void method(int x) {
2
3   ...
4 }
```

weave into →

```
1 void method(int x, int y) {
2   System.out.println("Msg: ");
3   ...
4 }
```

Strictly speaking, the weaving of the statement above is dynamic (since it transforms the behavior). However, this weaving can be achieved statically by altering the code structure. In general you are ill advised to think about dynamic crosscuts as static, because it takes away from the abstraction that is provided by the aspect (not illustrated in the above example). Nonetheless, we can perform useful and interesting crosscuts using a static method such as ISC. With the universalization of ISC we make the following observations:

- In a grammar-driven approach we can quickly introduce aspect-oriented notions to different languages. Admittedly not to the level of detail at which a tailored system such as ASPECTJ operates. But enough concepts can be introduced (or simulated) to perform a feasibility study for an aspect-oriented approach for a particular language.
- Through component model specifications we can quickly alter the assumed join point model. Hence, we can be allowed to more easily experiment with different restrictions on the considered join point model. The join point model is essentially defined by which implicit variation points are allowed. The actual selection of join points is done by traversing ASTs. Weaving is done by transforming ASTs using the ISC algebra.

  - We do not automatically have context information for join points as in AspectJ. Instead the generic mechanisms provided in a U-ISC–based composition system must be deployed to traverse the involved ASTs to retrieve the context information (this is demonstrated in Section 3.5.2, p. 91).
  - Fragment composition libraries can be developed that make exposed join points easier to select for end-users. This is also demonstrated in Section 3.5.2, p. 91.

  In conclusion, the AOP capabilities of ISC have been universalized along with the universalization of ISC. This holds promise for interesting prototyping of aspect-oriented ideas for different languages.

- *Syntactic units.* Our notion of fragment components is comparable to the notion of *syntactic units* presented in [64]. Syntactic units are arranged in *syntactic unit trees* that can be likened to composition programs. In this approach, so-called extension spots can be defined as alternatives for any fragment of code derivable from a nonterminal. Compared to our approach, there is no formalization of language extensions which allows for tailored extension of a language (to only allow the desired amount of variability). Furthermore, only explicit variation points are considered. Syntactic unit trees are mainly envisioned to be used in product-line engineering.

There are other software composition approaches that can be mentioned, for example Hyperspaces [69], CaesarJ [48], ObjectTeams/J [43] and others. However, many of these approaches are focused on separation of concern for object-oriented languages. We believe that at least a subset of such approaches can be realized using our composition technology. However, these approaches are not really related work as concerns our composition framework, hence we do not further comment on them here.

8 Outlook

In this chapter we discuss two possible extensions to our composition framework, and outline the initial investigations in these directions. First, in Section 8.1, we investigate the possibility of reusing component-oriented language extensions. Then, in Section 8.2, we discuss the possibilities of improving composition safety for specific composition systems.

8.1 Reusable language extensions

We are interested in component-oriented language extensions for DSLs, or languages in general in need of modularization capabilities. Two examples that we have discussed in this thesis are modules for *RL*, and modules for Xcerpt (cf. Section 4.5 and Chapter 5, respectively). In this section we will discuss the possibility of reusing such common language extensions—e.g. modules—for different languages.

Both these languages are rule-based in the style of logic-programming [68], and programs consist of sets of *rules*. As we have discussed, languages like these can benefit from support of reusable *modules*, which can also be defined as sets of rules. Simple modules for each of the languages are shown in Listings 8.1 and 8.2. Both modules are conceptually identical, but specified in their own languages. Both define two workers named `steve` and `marco`, and each module defines a rule stating that "workers are employees."

By studying the modules in Listings 8.1 and 8.2—and recalling our previous discussions about modules for rule-based languages—we can draw the following basic conclusions about what is needed to define modules:

MD1 *Module Naming.* Modules should be given names, or identifiers (`sales` is used for this in Listings 8.1 and 8.2).

MD2 *Module Definition.* Modules collect sets of rules of the underlying language.

In order to make use of the modules in Listings 8.1 and 8.2 we would need to import them:

```
MODULE sales

employee(X) :-
       worker(X).


worker(steve).
worker(marco).
```

LISTING 8.1: *A RL module (`file:sales.mrl`).*

```
MODULE sales

CONSTRUCT employee [ var X ]
FROM      worker   [ var X ]
END

CONSTRUCT worker [ "steve" ] END
CONSTRUCT worker [ "marco" ] END
```

LISTING 8.2: *An Xcerpt module (`file:sales.mx`).*

```
IMPORT file:sales.mrl

bonus(X, 100) :-
       employee(X).

```

LISTING 8.3: *Importing a RL module.*

```
IMPORT file:sales.mx

CONSTRUCT bonus    [ var X, "100" ]
FROM      employee [ var X ]
END
```

LISTING 8.4: *Importing an Xcerpt module.*

MI1 *Module Import.* It should be possible to import modules from programs or other modules.

An example of how the modules in Listings 8.1 and 8.2 can be imported and used is shown in Listings 8.3 and 8.4.

However, we recall that we have also been discussing important properties associated with modules, primarily *encapsulation*. Hence, we would like to refine our modules to the ones in Listings 8.5 and 8.6 (different syntax is used for the different modules, but their meaning is the same). Now, only the "employees" are accessible from the outside, and not the "workers" directly. Once we are able to encapsulate parts of modules, that is, define interfaces, we must be able to make use of those interfaces. For example, as shown in Listings 8.7 and 8.8, where the imported modules are appropriately queried. In addition, it might be necessary to provide data to a module for it to be able to perform its service (cf. Section 5.4, but not further discussed here). We add the following observations:

MD3 *Module Encapsulation.* It should be possible to encapsulate parts of modules

```
MODULE sales

@ employee(X) :-
       worker(X).


worker(steve).
worker(marco).
```

LISTING 8.5: *An encapsulated RL module (`file:sales2.mrl`).*

```
MODULE sales

CONSTRUCT public employee [ var X ]
FROM       worker [ var X ]
END

CONSTRUCT worker [ "steve" ] END
CONSTRUCT worker [ "marco" ] END
```

LISTING 8.6: *An encapsulated Xcerpt module (`file:sales2.mx`).*

```
IMPORT file:sales2.mrl AS sales

bonus(X, 100) :-
      IN sales ( employee(X) ).

```

LISTING 8.7: *Importing and querying an encapsulated* RL *module.*

```
IMPORT file:sales2.mx AS sales

CONSTRUCT bonus    [ var X, "100" ]
FROM IN sales ( employee [ var X ] )
END
```

LISTING 8.8: *Importing and querying an encapsulated Xcerpt module.*

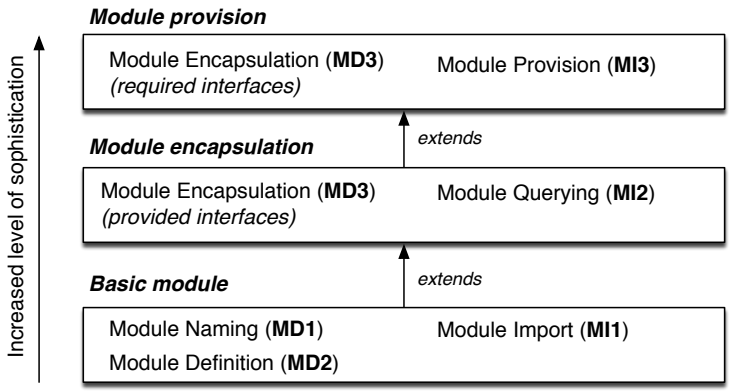


FIGURE 8.1: *We can separate the module concept for rule languages into three levels of sophistication:* basic module, module encapsulation *and* module provision.

by defining *provided* and *required* interfaces that define how the modules can be used.

MI2 *Module Querying.* It should be possible to make use of provided module interfaces, hence to query modules appropriately.

MI3 *Module Provision.* It should be possible to make use of required module interfaces, hence to provide module with data appropriately.

By studying the above module requirements, and recognizing the module definition and deployment similarities between the two languages, we make two observations: (1) There are different levels of module sophistication that can be deployed, and hence different sophistication of the involved language extensions. (2) Due to the similarities of how modules are defined and used for *RL* and Xcerpt, the "module extensions" can be defined in an abstract manner and reused for both, and possibly other, languages. We will elaborate on these observations below:

1. *Module sophistication levels.* For the module concept we have discussed, we define three levels of module sophistication, illustrated in Figure 8.1:

   (a) *Basic module (MD1–2, MI1).* The most basic module concept is to have the possibility to collect a set of rules as a module (using some intuitive syntax), give the module a name or identifier, and be able to import that module from a program or another module.

```
1  extends file:<basegrammar>.gr @ x as file:<reusegrammar>.gr .
2
3  % i) passive syntax
4  Module            =  moduleName:x.<identifier>, moduleStmt:x.<statement>* .
5  Module            <> x.<unit> .
6
7  OutInterface      =  interface:x.<out-interface> .
8  OutInterface      <> x.<out-interface> .
9
10 % ii) active syntax
11 ImportAs          =  moduleLocation:Module [ @Location ],
12                      moduleName:x.<identifier> .
13 ImportAs          -> @Composer .
14 ImportAs          <> x.<statement> .
15
16 InModule          =  moduleName:x.<identifier>, interface:x.<reference> .
17 InModule          -> @Composer .
18 InModule          <> x.<reference> .
```

LISTING 8.9: *Generic* C_mSL^+ *specification defining modules.*

(b) *Module encapsulation (MD1–3,MI1–2).* Beyond simply being able to define modules as reusable units, we can consider the possibility of encapsulating them. The most basic requirement is to be able to define *provided* module interfaces, and means of making use of such interfaces.

(c) *Module provision (MD1–3, MI1–3).* We can also allow to define *required* module interfaces, and enable constructs for providing the required data, depending on the particular module's functionality.[1]

2. *Reusing module extensions.* An interesting question to consider is if our discussed module extensions themselves can be defined as components – abstract "extension components" that can be reused across base languages. That is, if we can compose a "module extension" component with a base language, resulting in an augmented language with a module concept. Such abstract extension components could cover any of the module sophistication levels discussed above. We analyze the prospects of this idea by separating the problem into Krueger's distinction between abstraction *specification* and *realization* [60]:

   (a) *Reusing abstraction specifications.* To be able to specify the module abstraction we must define the suitable language extension. Consider the generalized C_mSL^+ specification in Listing 8.9. Each underlined and italicized construct enclosed in < and > belongs to a metalanguage (or can be seen as slots in a concrete setting). This specification corresponds to the "module encapsulation" sophistication level (the `fragtypes` construct has intentionally been left out for sake of simplicity). Both *RL* and Xcerpt uses the exact same specification to achieve module encapsulation. By parametrizing the specification for a particular language (grammar), the proper component

[1] We have defined module provision as a more sophisticated concept than module querying. This decision can be debated. In Xcerpt's case we mainly use modules without goal rules. Module provision only makes sense without module querying if the module in question contains at least one goal rule. To avoid goal rules in modules and for sake of simplicity, we have settled on this layering scheme here.

```
public I<statement> simpleImport(IModule module)
{
	I<statement> stmts = new I<statement>Impl();
	module.accept(new <basegrammar>Visitor() {
		public boolean visit(I<statement> stmt) {
			// collect statements
			getParamFragment().extend(stmt);
			return true;
		}
	});
	return stmts;
}
```

LISTING 8.10: *Simple generic template for extracting statements from a module.*

model specification can be achieved. Let us do this for Xcerpt:

$$
\begin{array}{lcl}
\texttt{basegrammar} & \longleftarrow & \texttt{xcerpt} \\
\texttt{reusegrammar} & \longleftarrow & \texttt{rxcerpt} \\
\texttt{identifier} & \longleftarrow & \texttt{Name} \\
\texttt{statement} & \longleftarrow & \texttt{XceptStatement} \\
\texttt{unit} & \longleftarrow & \texttt{XcerptProgram} \\
\texttt{out-interface} & \longleftarrow & \texttt{ConstructTerm} \\
\texttt{reference} & \longleftarrow & \texttt{QueryTerm}
\end{array}
\tag{8.1}
$$

Performing the bindings in (8.1) wrt. Listing 8.9 essentially say the following: (i) Our base language is Xcerpt; (ii) Use the `Name` construct from Xcerpt to represent module identifiers; (iii) Module statements are Xcerpt statements; (iv) Modules are on equal terms to Xcerpt programs; (v) Provided interfaces in modules will be defined on construct terms; and (vi) Query terms will be used when querying a module. By doing these substitutions, we will have a valid C_mSL^+ component models specification for Modular Xcerpt (at the "module encapsulation" level). The "basic module" level is even simpler, while the "module provision" level only requires two additional grammar definitions.

(b) *Reusing abstraction realizations.* But we also have to define the composition semantics for the parameterized C_mSL^+ specifications. Hence, we need to write the composition operators. The question is if it is possible to write reusable composition semantics. In the same way as the abstraction specifications can have different levels of sophistication, so can the abstraction realizations. Essentially, the implementation of the abstraction realization should correspond to the used abstraction specification.

In our approach, the required composition operators are implemented using generated core composition languages (Java APIs) and Java as the underlying platform. Since there are different core composition languages for *RL* and Xcerpt, we need a way to abstract from the particulars of the Java types provided by their APIs. This can for example be done by template programming. The generalized Java method in Listing 8.10 represents the implementation of the `IMPORT` construct for the "basic module" sophistication level. It extracts all the module statements and returns them. An actual con-

crete implementation of the composition operator can be attained by using, for example, template engines such as STRINGTEMPLATE.[2] Some form of generic programming could also be used, but Java generics is not powerful enough to solve this particular example.[3] Other solutions are also possible. For example, having abstract base classes implementing the generic solutions, which can then be sub-classed to provide language-specific solutions. But, such investigations are left as future work.

The higher the sophistication of the module abstraction specification, the more tightly connected the realization becomes to the underlying language. That is, the more complex the encapsulation requirements are, the more tailored the solution becomes to a particular language. These reusable language extensions can in an abstract way be likened to object-oriented software frameworks. In lightweight framework use, many default settings can be left in place. However, for more heavyweight deployment, more settings have to be tailored for the particular usage scenario. This observation mirrors the usage of a reusable module extension: the "basic module" level can be achieved by simple parameterization of generic artifacts (e.g. Listings 8.9 and 8.10), while the "module provision" level requires a more specialized deployment.

Summary

We make the following concluding remarks:

- Language extensions that are aimed for abstraction specifications can be seen as components, and can hence be reused for different base languages.
- Generic component models, which include the abstraction specification structure (cf. Listing 8.9), can be parameterized by base languages (grammars) to achieve a particular component-oriented language extension.
- The implementation of the composition operators that realize the abstraction specification under consideration can be tightly connected to a particular base language, and is as such hard to get for free. However, for certain simple extensions, large parts of the implementation can be generated.
- Composing language extensions in this way can be seen as a special niche in the more general topic of language engineering or language composition.
- Investigating in more detail what component-oriented language extension "components" are seems to be an interesting research topic for the future.

8.2 Abstraction-specific composition contracts

As mentioned in Chapter 2, our presented composition framework only guarantees context-free syntactical correctness. That is, every final composition result is guaranteed to be a valid instance of the considered base language. Hence, we do not support

[2] http://www.stringtemplate.org/

[3] http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

```
1  MODULE books
2
3  CONTRACT ( Input: Books, Output: Titles, Typespec: file:contract.xts )
4
5  CONSTRUCT public titles [ title [ var X ] ]
6  FROM       internal {{ title [ var T ] }}
7  END
8
9  CONSTRUCT internal [ all title [ var T ] ]
10 FROM       public books [ book [ title [ var T ], author [ var A ] ] ]
11 END
```

LISTING 8.11: *An Xcerpt module with a* contract *dictating its expected input data, and promised output data.*

context-sensitive syntactical correctness, that is, conditions specified in the static semantics of a language. Context-sensitive syntactical correctness is dependent on the particular language in which the fragments are written, which makes it hard to deal with in a generic manner. However, what if we *would* like to support such guarantees for a particular language and composition system? But, even if we want to take static semantics for a particular language into account, from a fragment composition perspective it is unclear when to perform the checks. This should ideally be done at every low-level composition step (e.g. when executing *bind()* or *extend()*). However, the considered static semantics might not be well-defined for every low-level composition step. Performing checks on the final composition result is still possible, but without adequate tracing mechanisms it is difficult to identify the origin of any errors.

With the introduction of E-ISC (cf. Chapter 4) we enabled abstractions that do not have to be seen as 'fragments.' An example of such an abstraction is the 'module' for Xcerpt (cf. Chapter 5). If it is difficult to consider static semantics on the level of fragments, is it possible to define other kinds of safety guarantees on these new abstractions, e.g. Xcerpt modules? In other words, the question is if it possible to define additional composition conditions on abstractions defined in an E-ISC–based approach – on the level of extended languages semantics, rather than on the level of their transformations. We have investigated this for Xcerpt modules, and present the idea here.

The idea is to use existing typing technology for Xcerpt to ensure the correct definition and deployment of Xcerpt modules. One such typing technology for Xcerpt is presented in [95]. The type checker XCERPTT is based on the theoretical work presented in [95].[4] We here give a brief understanding on how this typing technology can be used. The reader is directed to [95] for more details on typing for Xcerpt. Let Q represent a query, C a construct term in a construct rule, and G a construct term in a goal rule. We denote $C \leftarrow Q$ a construct rule, $G \leftarrow Q$ a goal rule, and $C \leftarrow in[r, Q]$ a construct rule querying the external resource r. Assume we have a type specification T, an input type I of T and an output type O of T. Given the Xcerpt program:

$$G_1 \leftarrow Q_1, \ldots, C_n \leftarrow in[r, Q_n] \tag{8.2}$$

the type checker can associate the type I with the resource r, and the output type O with the construct term G_1 and check if the program is type correct. If the type checker

[4] http://www.ida.liu.se/~artwi/XcerptT

says that the program is type correct, then this means the following: if the input data is of type I, then the output is of type O.

We can deploy this typing technology to improve Modular Xcerpt composition robustness. This is done by associating a *contract* with each module. This can look like in Listing 8.11, where the contract consists of a *type specification* (`file:contract.xts`), and an *input* and *output type* from the type specification. The type specification is an XML Schema or DTD (Document Type Definition) document (i.e. a description of the valid structure of XML documents). In the context of modules, the type specification is used to specify the structure of the expected module input and output data. The precise input and output types are specified in the contract (in Listing 8.11, `Books` and `Titles`, respectively). Notice that such a contract is specified on the level of the module, not on the individual low-level composition steps that realize the module concept. Hence, on the level of the extended language semantics, not on its compositional semantics.

Given a module with a contract (cf. Listing 8.11), we can use type verification to check the correct definition and deployment of modules. This would be done in two steps:

1. *Verify module wrt. its contract.* Assume the module m to be (for sake of simplicity we assume that no goal rules are present in the module):

$$\widehat{C_1} \leftarrow Q_1, \ldots, C_n \leftarrow \widehat{Q_n} \tag{8.3}$$

   Transform the module into:

$$G_1 \leftarrow Q_1, \ldots, C_n \leftarrow in[r, Q_n] \tag{8.4}$$

   where G_1 is the same construct term as C_1, and r is some dummy resource. Given the module m's contract type specification, the input type and output type, XCERPTT can now check type correctness of the program in (8.4) in the same way as for (8.2). If the type checker confirms type correctness for (8.4), we say that the module m (cf. (8.3)) is *valid* wrt. its contract. This means that: if the module input data is of the specified input type, then the module output data is of the specified output type. If the type checker does not confirm type correctness, there might be a problem with the module and the module developer could be informed.

2. *Verify module usage.* Let us assume that module m with associated contract (T, I, O) is valid wrt. its contract, where T is the type specification and I and O are the input and output types, respectively. Now, suppose we have the following program, where $m(Q_1)$ means that module m is being queried:

$$G_1 \leftarrow m(Q_1) \tag{8.5}$$

   If we have a type specification T', and a type O' of T' for the output of (8.5), we can check the usage of the module by first transforming the program into:

$$G_1 \leftarrow in[r, Q_1] \tag{8.6}$$

   If we then associate the type O of m's contract with resource r, and O' with the result G_1, we can have the type checker verify the type correctness of the

program in (8.6). A positive answer from the type checker would mean that the module usage in (8.5) is *valid* wrt. *m*. This says: under the assumption that the data provided to the module *m* is of input type *I* of the module's contract, the result of the composed program is of the expected type (O'). We can use the same method for checking module provisions (i.e. when data is provided to a module). If we do not have the type specification for (8.5), we can have the type checker infer the resulting type, which might give hints as to whether or not the module was used correctly.

The above has described the basic idea of a method for providing and verifying interface contracts on the level of E-ISC–specified abstractions (components). This can be contrasted to the inability of providing any meaningful context-sensitive safety conditions on the level of U-GBM–based, or U-ISC–based, fragments.

Summary

We make the following concluding remarks:

- It is important to be able to provide safety guarantees that go beyond context-free syntactical correctness. Being able to do so will improve composition robustness. It is not valuable to compose programs that cannot be executed or used as expected.
- Specifying deployment conditions on E-ISC–based abstractions is a means to improve the quality of composition results in our framework.
- Contracts on components, as demonstrated in Listing 8.11, cannot only help in improving composition robustness, but can also be used to provide useful documentation for such components. For example, the contract in Listing 8.11 essentially explains how the module is expected to be used.
- One drawback with the example presented above is the assumption that there exists an external tool that can be deployed for checking the contracts. For other languages, this might not be the case. Some other languages might also have type systems, or other useful mechanisms, but not a stand-alone tool that can be deployed in this manner.
- More research should be conducted to investigate what other possibilities exists for improving composition robustness. Being able to provide improved guarantees on composition results is essential for the success for any composition approach.

9 Conclusions

We have in the course of this thesis presented a fragment-based composition framework, and two instantiations of the same. The composition framework does not commit to any particular underlying language, while the two instantiations pertain to declarative languages often found on the Semantic Web. The two addressed languages are the rule-based query language Xcerpt [77] and the DL-based ontology language OWL [73]. The framework itself builds upon previous work in the field of fragment-based composition, namely grammar-based modularization (GBM) [63] and invasive software composition (ISC) [5]. We have *universalized* these approaches, which means that we have investigated and described how they can be applied to arbitrary context-free languages, based on their grammars. Hence, our universalization results in a *grammar-driven* composition approach. The concrete result of this is first universal GBM (U-GBM), and then universal ISC (U-ISC) which builds upon and extends U-GBM. Then, on top of these generic techniques, we defined *embedded* ISC (E-ISC) as an approach for applying the composition technique for a particular category of domain-specific languages (DSLs), namely *non-embedded* DSLs (NE-DSLs). However, the approach is not limited to such languages.

In summarizing the presented work, and putting it in perspective, we will relate it to two existing, popular and common software engineering disciplines and approaches: aspect-oriented programming (AOP) and DSL embedment techniques. However, we will not make a comparison on a technical level, but rather on a motivational level. In comparing with AOP techniques we will motivate and highlight the importance of the need to move from U-ISC to E-ISC. And in comparing with DSL embedment techniques we will distinguish our approach from other related approaches and see how we identify a previously largely unaddressed problem.

1. *AOP techniques.* AOP approaches, such as ASPECTJ, allow to weave in modularized code (the aspects) into a base program such that the overall system objective is realized. The aspects are typically written in an extended language (e.g. in ASPECTJ, which is an extension of Java). The actual weaving can take place on the source code level, byte code level, or binary code level. Whatever the choice,

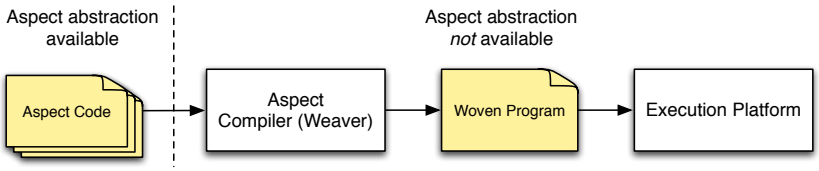


FIGURE 9.1: *Aspect systems can transform programs on different levels: source code, byte code, or binary code level. Regardless of the particular level of transformation, it is a transformed program that is being executed in the end.*

these are really technical details managed by the aspect system, or its compiler. In the end, it is a transformed program that is being executed. This is illustrated in Figure 9.1. Being aware of these transformations can be a good way of understanding what happens under the hood when using as aspect system. However, proponents of AOP techniques, in particular ASPECTJ, warn programmers from thinking in terms of the *transformations* (see e.g. [61, p. 42]). The reason for this is clear. The purpose of an aspect is to capture an abstraction. If programmers think in terms of how that abstraction is technically realized (e.g. how the call stack is modified), the abstraction is partly lost and no longer fulfills its purpose. Instead, aspect programmers are encouraged to think on the level of aspects in the language extension provided for them – to think in the extended language semantics.

In U-ISC there are no extended languages to think in terms of, hence no extended language semantics. There are only source code *transformations* to think about, in direct contrast to what we have learned from systems such as ASPECTJ. In this sense, there are no real abstractions. Fragments with slots, or boxes with hooks in COMPOST, *can*, as previously explained, be seen as abstractions. But we argue that such abstractions are not directly useful or desirable by programmers. Hence, there is clearly a need for an approach such as E-ISC to be layered on top of U-ISC.

2. *DSL embedment techniques.* There have been many approaches developed for embedding DSLs into more general-purpose host languages (GPLs), e.g. Java. Some of the approaches are [12, 17, 45, 90]. The motivation for such embedments is clear and such works are important for providing developers with better, more integrated and appropriate languages. The lack of appropriate and domain-specific syntax is a problem for a wide range of GPLs. Acknowledging this, some approaches do not commit to a particular host language (e.g. [17]).

   The core problem we are addressing closely mirrors the above-described problem, but is fundamentally different: We essentially aim at embedding non–domain-specific abstractions into DSLs, in particular NE-DSLs. This differentiation is illustrated in Figure 9.2.

   We are, to the best of our knowledge, not aware of other approaches that clearly have identified this problem, in particular in a general setting.[1] Individual languages always evolve and are extended with new features, for example with new

[1] An approach with a closely related goal is Hyperspace, but mainly aimed at augmenting GPLs with different kinds of concern separations, such as aspects, roles, views etc. [69].

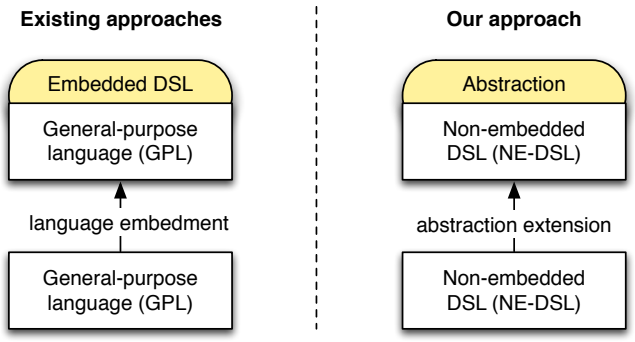


FIGURE 9.2: *There exist approaches for embedding DSLs in general-purpose languages. In contrast, we aim for extending DSLs with new abstractions.*

abilities for component-based development. For example, there is a current interest in modularization constructs for ontology languages (cf. Section 6.5, p. 192). However, this problem is always addressed in isolation for each individual language. Our goal is to provide a general method for achieving the same. In the same way that [17] does not commit to a particular host language for DSL embedment, we do not commit to a base language to be extended.

We believe that this thesis has identified a previously unidentified and unaddressed problem: the need for a general method and technique for enabling new abstractions for a range of different languages.

In this thesis we worked in recognition of the above-described issues. We have enabled the usage of ISC-based techniques that moves away from only having to consider fragment *transformations*. We demonstrated two component-oriented language extensions for two different DSLs, and showed how it is possible to allow for end-users to think and design in terms of the extended language semantics:

1. We provided modules for Xcerpt (cf. Chapter 5) where query programmers can work with modules and their provided and required interfaces.

2. We provided role models for (Manchester) OWL (cf. Chapter 6) where ontology designers can work with role models and their role type interfaces.

For each of these extensions, their realization uses a well-designed *union* composition operator that respects the involved interfaces. For both cases, the involved components (modules and role models) are merged with the base programs, but where the merging is carefully defined in such a way that the interfaces are used and respected. From a technical viewpoint, the merging is achieved via composition. By using a composition technique that reduces programs of the extended language into programs of the base language, we are able to reuse existing tools for the extended languages. This is a very important benefit of our approach. Hence, it is assumed that users never work on composition results. The composition results are instead directly sent to the involved tools (compilers, interpreters, reasoners etc.).

We believe that *union* composition operators are in particular applicable to the declarative languages we have addressed. If this is the case, there is promise for further

applications in the area of the Semantic Web, where declarative languages are prominent.

We make the following concluding remarks:

- In contrast to program transformation techniques, a composition technique must have a well-defined *component model*. One of the most important ideas presented by U-ISC and E-ISC–based techniques is the flexibility in how the underlying component model can be changed. Being able to quickly modify the component model allows for experimentation and prototyping. Hence, these techniques can be used to experiment with composition and abstraction ideas for different languages. Such opportunities should be further exploited in the future. Targeting the composition framework for such scenarios is an interesting way forward.

- A drawback with easily modifiable component models is that the approach can be considered *fragile*. Not only does the implementation of composition operators or fragment composition libraries already depend on a base language which might be subject to evolution, but also on the precise component model specification. Hence, developers must take care not to invalidate large code bases that have *committed* to a base language grammar and component model specification. This *fragility property* of the composition framework might hint at its suitability for experimentation, and not for stable composition system environments.

- We discussed a realistic composition system development process in Section 4.6 (p. 124). Our experience using our composition technology has taught us that most of the development time goes into the "requirement phase." That is, much time is spent considering what abstraction is describable for a particular language. With this knowledge, it should be clear that the work going into trying to define *reusable language extensions* is indeed important future work. We discussed initial ideas in Section 8.1 (p. 211).

- It is clear that it is possible to develop more powerful component approaches for DSLs by committing to a particular base language. That is, tailored solutions are always more powerful. However, for such tailored solutions, there are costs involved, e.g. tool construction. We believe that a lightweight approach is often sufficient for increased programmer productivity. Such a lightweight approach, using our framework, can be achieved at minimal cost.

- A limitation of our approach is that we only consider context-free languages. This means that we do not give context-sensitive safety guarantees during composition. This makes it possible to compose programs that, even though syntactically valid, will not work as expected. Some errors that can give cause to unexpected problems can be easy to find, but there is currently no way of finding them and informing the user. However, there are some ideas to address this for E-ISC–based compositions (see Section 8.2, p. 216). Some context-sensitive issues can be found by the underlying tools (e.g. an interpreter), but such errors will be found too late and it is hard to provide useful error messages to the users. Better error handling for fragment-based compositions is certainly an important future research task.

- In Section 2 we defined our safety conditions wrt. the languages that CFGs generate. This was quite a natural choice since this interpretation of CFGs is

the de facto standard. However, other approaches are certainly possible. One possibility is to instead define the safety conditions wrt. derivation trees. Such derivation trees retain the underlying grammar structure. This might lead to more practical approaches for statically calculating grammar types and verifying safety conditions. However, this has currently not been investigated. Different formalizations of the composition techniques could be valuable to have. This should be considered part of future work.

Bibliography

[1] Extended BNF. ISO Standard, 13 August 2001. Available at `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26153`.

[2] The AspectJ Project, October 2008. Available at `http://www.eclipse.org/aspectj/`.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference, 1972.

[5] U. Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[6] U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and J. Johannes. Modular Web Queries—From Rules to Stores. *3rd International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS'07). Vilamoura, Algarve, Portugal, Nov 27, 2007*, 4806/2007:1165–1175, 2007.

[7] U. Aßmann, S. Zschaler, and G. Wagner. *Ontologies, Meta-Models, and the Model-Driven Paradigm*. Springer, 2006.

[8] F. Baader, D. Calvanese, and D. McGuiness(et.al.), editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[9] C. W. Bachman and M. Daya. The role concept in data models. In *VLDB '1977: Proceedings of the third international conference on Very large data bases*, pages 464–476. VLDB Endowment, 1977.

[10] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. Number 3564, pages 35–133. Springer-Verlag, 2005.

[11] J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281, New York, NY, USA, 2002. ACM.

[12] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In *In Proceedings of the 5th International Conference on Software Reuse*, pages 143–153. IEEE, 1998.

[13] S. Boag, D. Chamberlin, et al. XQuery 1.0: An XML Query Language. W3C Recommendation, 23 January 2007. Available at `http://www.w3.org/TR/xquery/`.

[14] D. Box and A. Hejlsberg. LINQ: .NET Language-Integrated Query, 2008. Available at http://msdn.microsoft.com/en-us/library/bb308959.aspx. Accessed 1 October 2008.

[15] C. Brabrand and M. I. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. In *Proceedings ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'02*, pages 31–40. ACM, 2002.

[16] C. Brabrand and M. I. Schwartzbach. The metafront system: Safe and extensible parsing and transformation. *Science of Computer Programming*, 68(1):2–20, 2007.

[17] M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM Press.

[18] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, 10 February 2004. Available at http://www.w3.org/TR/rdf-schema/.

[19] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1361–1398, July 1994.

[20] F. Bry, T. Furche, and S. Schaffert. Initial Draft of a Language Syntax. Technical Report IST506779/Munich/I4-D6/D/PU/a1, Institute for Informatics, University of Munich, 2006.

[21] F. Bry and S. Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Sardinia, Italy (14th June 2002)*, 2002.

[22] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 295–310, London, UK, 2003. Springer-Verlag.

[23] J. Cardoso. The Semantic Web Vision: Where Are We? *Intelligent Systems*, 22(5):84–88, 2007.

[24] J. Clark. XSL transformations (XSLT). WWW Page, November 1999. Available at http://www.w3.org/TR/xslt.

[25] B. Cuenca Grau, Y. Kazakov, I. Horrocks, and U. Sattler. A Logical Framework for Modular Integration of Ontologies. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 298–303, 2007.

[26] B. Cuenca-Grau and B. Motik. OWL 2 Web Ontology Language: Model-theoretic semantics. W3C Working Draft, 2008. Available at http://www.w3.org/TR/owl2-semantics/.

[27] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM.

[28] A. Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[29] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[30] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical Report 01.12, 2000.

[31] M. Flatt. Composable and compilable macros – you want it when? *SIGPLAN Not.*, 37(9):72–83, 2002.

[32] M. Fowler. MF Bliki: DomainSpecificLanguage. WWW Page, July 2008. Available at `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`. Accessed 2 July 2008.

[33] P. Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[34] B. C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Modularity and Web Ontologies. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *Proceedings of KR2006: the 20th International Conference on Principles of Knowledge Representation and Reasoning, Lake District, UK, June 2–5, 2006*, pages 198–209. AAAI Press, 2006.

[35] T. Gruber. What is an ontology? WWW Page, 1992. Available at `http://www-ksl.stanford.edu/kst/what-is-an-ontology.html`. Accessed 28 September 2008.

[36] N. Guarino. Concepts, attributes and arbitrary relations - Some linguistic and ontological criteria for structuring knowledge bases. *Data and Knowledge Engineering*, 8(3):249–261, 1992.

[37] N. Guarino and C. A. Welty. Evaluating ontological decisions with OntoClean. *Communications of the ACM*, 45(2):61–65, 2002.

[38] G. Guizzardi. *Ontological Foundations for Structural Conceptual Models*. PhD thesis, Universiteit Twente, Netherlands, 2005.

[39] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On Language-Independent Model Modularisation. *Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE*, 2008. To Appear.

[40] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect Orientation for Your Language of Choice. In *In Proceedings of Workshop on Aspect-Oriented Modeling at MoDELS 2007*, 2007.

[41] J. Henriksson, M. Pradel, S. Zschaler, and J. Z. Pan. Ontology Design and Reuse with Conceptual Roles. In *The Second International Conference on Web Reasoning and Rule Systems (RR'08) (to appear)*, Lecture Notes in Computer Science. Springer, 2008.

[42] S. Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In *In Proceedings of Net Object Days 2002*, pages 248–264. Springer, 2002.

[43] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264, London, UK, 2003. Springer-Verlag.

[44] S. Herrmann. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[45] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In *To appear in Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*. ACM Press, 2008.

[46] M. Horridge and P. F. Patel-Schneider. Manchester Syntax for OWL 1.1. In *International Workshop OWL: Experiences and Directions (OWLED '08)*, 2008.

[47] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196–196, 1996.

[48] M. M. Ivica Aracic, Vaidas Gasiunas and K. Ostermann. Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I.*, volume 3880, pages 135–173. Springer, February 2006.

[49] J. Johannes. Complex Composition Operators for Semantic Web Languages. Diploma Thesis, 2006.

[50] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, 1st edition*. Prentice Hall, 1978.

[51] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[52] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams, Indianapolis, IN, USA, 2002.

[53] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[54] P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[55] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/rdf-concepts/`.

[56] D. C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[57] B. B. Kristensen, O. L. Madsen, and B. M. ller Pedersen. The when, why and why not of the BETA programming language. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 10–1–10–57, New York, NY, USA, 2007. ACM.

[58] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Abstraction mechanisms in the BETA programming language. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 1983. ACM.

[59] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Syntax Directed Program Modularization. In *Interactive Computing Systems (ed. P. Degano, E.Sandewall)*, 1983.

[60] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[61] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[62] H. Liu, C. Lutz, M. Milicic, and F. Wolter. Reasoning about Actions using Description Logics with general TBoxes. In *European Conference on Logics in Artificial Intelligence (JELIA '06)*, pages 266–279. Springer, 2006.

[63] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[64] M. Majkut and B. Franczyk. Generation of Implementations for the Model Driven Architecture with Syntactic Unit Trees. In *Proceedings of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.

[65] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[66] O. Nierstrasz and T. D. Meijler. Requirements for a Composition Language. In *ECOOP '94: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 147–161, London, UK, 1995. Springer-Verlag.

[67] I. Niles and A. Pease. Towards a standard upper ontology. In *International conference on Formal Ontology in Information Systems (FOIS '01)*, pages 2–9. ACM, 2001.

[68] U. Nilsson and J. Małuszyński. *Logic, Programming, and PROLOG*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[69] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[70] J. Z. Pan, L. Serafini, and Y. Zhao. Semantic Import: An Approach for Partial Ontology Reuse. In *Proc. of the ISWC2006 Workshop on Modular Ontologies (WoMO)*, 2006.

[71] T. Parr. Enforcing strict model-view separation in template engines. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM.

[72] T. Parr. ANTLR — ANother Tool for Language Recognition — Parser Generator. WWW Page, October 2008. Available at `http://www.antlr.org`.

[73] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/owl-semantics/`.

[74] M. Pradel, J. Henriksson, and U. Aßmann. A Good Role Model for Ontologies: Collaborations. *International Workshop on Semantic-Based Software Development. Co-located with OOPSLA'07, Montreal, Canada, Oct 22, 2007*, 2007.

[75] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Candidate recommendation, W3C, 2007.

[76] T. Reenskaug, P. Wold, and O. Lehne. *Working with Objects, The OOram Software Engineering Method*. Manning Publications Co, 1996.

[77] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Institute of Computer Science, LMU, Munich, 2004.

[78] S. Schaffert, F. Bry, and T. Fuche. Simulation Unification. Technical Report IST506779/Munich/I4-D5/D/PU/a1, Institute for Informatics, University of Munich, 2005.

[79] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In *Proceedings of 17th European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany, July 21-25, 2003*, Lecture Notes in Computer Science, pages 248–274. Springer, 2003.

[80] Y. Smaragdakis and D. S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *Software Engineering and Methodology*, 11(2):215–255, 2002.

[81] J. Sowa. *Conceptual structures: information processing in mind and machine*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[82] J. Sowa. *Using a lexicon of canonical graphs in a semantic interpreter*, pages 113–137. Cambridge University Press, New York, NY, USA, 1988.

[83] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*, 35(1):83–106, 2000.

[84] F. Steimann. The role data model revisited. *Applied Ontology*, 2(2):89–103, 2007.

[85] E. Sunagawa, K. Kozaki, Y. Kitamura, and R. Mizoguchi. Role Organization Model in Hozo. In *International Conference on Managing Knowledge in a World of Networks (EKAW)*, pages 67–81. Springer, 2006.

[86] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, NY, 1998.

[87] The COMPOST Consortium. The COMPOST system. WWW Page. Available at `http://www.the-compost-system.org`.

[88] The Gene Ontology Consortium. The Gene Ontology. WWW Page. Available at `http://www.geneontology.org/`.

[89] The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25(1):25–29, May 2000.

[90] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute Grammar-based Language Extensions for Java. In *European Conference on Object Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer Verlag, July 2007.

[91] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[92] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. In *Domain-Specific Program Generation*, volume Volume 3016/2004, pages 216–238. Springer Berlin / Heidelberg, 2004.

[93] P. Wegner. Varieties of Reusability. In *Proceedings of Workshop on Reusability in Programming*, pages 30–44, September 1983.

[94] D. Weise and R. Crew. Programmable syntax macros. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 156–165, New York, NY, USA, 1993. ACM.

[95] A. Wilk. *Types for XML with Application to Xcerpt*. PhD thesis, Linköping University, Sweden, 2008.

[96] E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. In *Proceedings of LDTA 2007, 7th Workshop on Language Descriptions, Tools, and Analysis*, 2007.