"A product-line for families of program translators : a grammar-based approach"

Ordonez Camacho, Diego

Abstract

The need for translating program source-code between many different programming languages arises in some domains for which many such languages coexist. One such domain is that of space-mission planning, where a family of operations languages exists: different space operators use different languages to capture the operational knowledge to test and to control spacecrafts. Building a program translator from a single source to a target language already requires considerable time and effort because of the inherent complexity of every step in the process. If in addition, there is a big family of many such languages in some application domain, the cost and effort of creating program translators between any of them becomes prohibitive. In this thesis we address this translation problem by combining several techniques to generate a family of program translators: a product-line approach provides the support for a reusable translator framework; a grammar convergence reverse-engineering approach...

Document type : *Thèse (Dissertation)*

A product-line for families of program translators. A grammar-based approach

Département d'Ingénierie Informatique
École Polytechnique de Louvain
Université catholique de Louvain

Dissertation

A product-line for families of program translators. A grammar-based approach

Diego Antonio Ordóñez Camacho

26th August 2010

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor in Engineering Sciences

Thesis Committee:

Prof. Olivier Bonaventure (Chair)	UCL, Belgium
Prof. Kim Mens (Promoter)	UCL, Belgium
Prof. Paul Klint	CWI, The Netherlands
Prof. Anthony Cleve	INRIA, France
Prof. Charles Pecheur	UCL, Belgium

A product-line for families of program translators. A grammar-based approach

Pôle d'Ingénierie Informatique
Institute for Information and Communication Technologies, Electronics and Applied Mathematics
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work has been supported by the APPAREIL project of the First Europe Objectif 3 program of the Direction Générale des Technologies de la Recherche et de l'Energie, Walloon Region Ministry, and the MoVES project of the Interuniversity Attraction Poles Program of the Belgian Science Policy, Belgian State.

To my family.

Acknowledgements

I am in debt with many people, in one way or another, for their help and support. I apologise beforehand with those that I am not mentioning here. Either way, mentioned or not, words will always be insufficient to express them how thankful I am.

I owe my deepest gratitude to my adviser, Kim Mens. He provided me with the opportunity, with guidance and with friendship. The patience he showed when confronted to my stubbornness is biblical, and the active help he gave me with my work deserves him another PhD. Thanks Kim. I am very lucky I was given the opportunity to know you and to work with you.

I thank all the jury members, Olivier Bonaventure, Anthony Cleve, Charles Pecheur and Paul Klint, for the time and effort invested in evaluating this work. Their feedback was of the utmost importance to improve the quality of the text.

Two very important persons for this work have been Mark van den Brand and Jurgen Vinju. When Paul Klint received me for an internship in CWI's SEN1 laboratory, Mark and Jurgen initiated me in the wonders of program transformation. We wrote together a couple of papers, and Mark was part of my follow-up committee. I thank them and the rest of the CWI team with whom I had the opportunity to share and work.

To the members of the INGI department, thank you. I had the great opportunity to have them as teachers, colleagues and friends. I would especially like to thank Stéphanie Landrain and Viviane Dehut. They helped me whenever they could, always with a smile.

The RELEASeD laboratory will always be close to my heart. Me and my closest friends and colleagues have worked there: Sebastián González, Alfredo Cádiz, Nicolas Cardozo, Sergio Castro, Johan Brichau, Angela Lozano, and Kim Mens. They have not only contributed to my work, but also to my life.

I am grateful to RHEA Systems and all the people I met there. They also made possible this work.

To all my friends, thank you. They are my extended family, and I have always received from them more than I gave. I am looking forward for the opportunity to get even.

Thanks to my beloved family. To my parents Oswaldo and Marina. They would be proud. To my brother Fernando and my sister Marina for caring about me. To Verónica for her support no matter how rough the weather was. And finally, thanks to my daughter AnaClara for being there and brighten up my life.

Contents

List of Figures

1 Introduction

The need for translating program source-code between many different programming languages arises in some domains for which many such languages coexist. One such domain is that of space-mission planning, where a family of operations languages exists: different space operators use different languages to capture the operational knowledge to test and to control spacecrafts.

Building a program translator from a single source to a target language already requires considerable time and effort because of the inherent complexity of every step in the process. If in addition, there is a big family of many such languages in some application domain, the cost and effort of creating program translators between any of them becomes prohibitive.

In this thesis we address this translation problem by combining several techniques to generate a family of program translators: a product-line approach provides the support for a reusable translator framework; a grammar convergence reverse-engineering approach enables to extract common models from programming languages and programs, and a language-parametric grammarware approach provides the specific translation and transformation techniques.

1.1 Context

We were first confronted with the problem of program translation in the context of a research project with a company, RHEA Systems, specialised in the domain of space-mission planning. The company has a software suite that, amongst other functionalities, provides support for designing space operations procedures. These procedures are programs for executing specific operations during a space mission, such as reorienting an antenna, or correcting the satellite's orbit. Although the software suite is being successfully used by different mission control centres around the world for designing new operations procedures, there is one additional requirement demanding special attention. This new requirement consists of being able to import into the software suite existing operations procedures, created by external means, to check or modify them thanks to the suite's graphical interface.

This requirement was solved at first by the company by implementing specific importers, or translators, but soon this solution became impractical. The company's client database started to grow and, given that many

of their clients were using their own set of languages to program their procedures, the number of specific translators required started to grow quickly. Moreover, new versions of old languages appeared, and completley new languages started to be proposed as alternatives or as standards. To complete the picture, some of the mission control centres started to closely collaborate with each other. The translation problem became too big to keep working under the same old schema. The development time and effort for each of the specific translators increased considerably, and a different solution had to be considered.

To program operations procedures, a specific kind of programming languages is used. These languages are called *operations languages*, because they are used to implement the concrete operations that have to be completed in the context of a space mission. Most operations languages are very similar. They are imperative languages, inspired by older languages like ADA or Pascal, whose functionality is focused on (testing or) controlling when certain functions or commands are invoked for execution on another equipment (in particular, on a satellite). Operations languages provide the means to refer to the different elements in the space system, like activity commands or reporting data, and the means to define the procedural script to interact with these elements. Figure 1.1 presents a schematic view of this interaction.

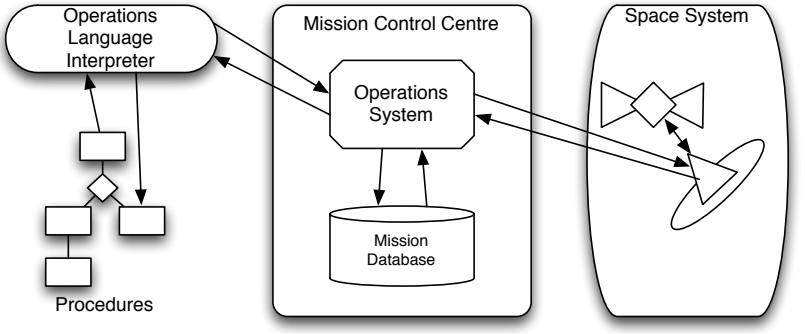


Figure 1.1: The Space System Interaction.

Operations languages procedures are executed by an interpreter attached to the mission control centre. The interpreter runs the program, until it gets to an instruction requiring communication with the satellite. The instruction is then sent to the control centre operation system. In the control centre there is a mission-specific database that contains all the instructions that the satellite can execute. The mission database allows the control centre to verify the instruction and its parameters, to translate the information in terms of a low-level command invocation, and to send this command to the satellite. Similarly, any information coming from the satellite is coordinated by the control centre, who receives the information, and stores it in the

mission database to make it accessible to the interpreter. It is the mission control centre who is responsible for all the required communication between operators and the satellite. With this in mind, we can see the execution of operations procedures by the mission control centre as an automated version of human interpreters, which is how the operations languages historically came into existence.

Even though many operations languages are as powerful as any general-purpose programming language. In practice though, they are used as domain-specific languages, exploiting only the functionality required to invoke the commands in the control centre, and to control the flow of the execution of these commands.

Regardless of certain minor differences, at the moment of our analysis a rather traditional technique was being used by the company to build the translators used to import procedures from operations languages. The technique mainly involved an attributed grammar parser, and the translators thus generated presented an important drawback: the code to generate the translated procedure ends up embedded into the grammar used to parse the original procedure, which leads to a lack of modularity, making reuse (towards the languages) very difficult and limited. The different parts of the translators are too specific to the source and target languages.

Although in our work we focused mainly on this language translation problem in the domain of operations languages, we have good reasons to believe that this translation problem is not restricted to the domain of operations languages only. For instance Cleve and Hainaut [28] present a case where data and programs from a legacy database needed to migrate to other more contemporaneous database. The languages used to program the queries and transformations for any of these databases can be considered as a *languages family*. For those cases where a migration between two relational database platforms was necessary, a program translation approach could have been used at some extent, as an alternative to some of the program transformation processes implemented.

1.2 Problem and Thesis Statement.

A typical program translator performs a series of large, non-trivial steps, to transform programs from a source to a target language. Typical steps are lexical, syntactic and semantic analysis, tree transformation and term rewriting, unparsing and verification. Each of these steps have an inherent complexity degree, and therefore the process of building a full-fledged translator can be considered as highly complicated and time-consuming.

The difficulty of building such translators is aggravated in domains where many similar languages coexist and where, to translate programs from any such language to any other, a family of translators must be built. If in the worst case we need to provide a translator for every possible combination of languages in the domain, we end up with a number of translators that is

quadratic in the number of languages in the family. A substantial reduction in the number of required translators can be achieved if we use a pivot language to and from which we translate all other languages. But even with such solution, the number of translators needed still remains important. Moreover, since a large portion of these translators have many commonalities, the per-translator approach leads to duplicated efforts.

As we will see in Section 3.2, many alternative techniques exist to build program translators like, for instance, Attribute Grammars [132] or Term Rewriting [8]. These techniques have been thoroughly explored and each has its own strengths and weaknesses. The purpose of this work is not to propose yet another technique for program translation, but rather to use the best of what has been proposed so far, to build a framework that can produce a large number of program translators, with a minimal programming effort, and with confidence in the result of the translation.

The solution we propose explicitly takes into account that we work with a family of languages. After all, the problem is not about building a translator, but about building families of translators. We need to think in terms of commonalities and differences between different languages and language constructs, and we need to strive for reuse at every possible level. We do not just need many translators, but we need them fast and we need to respond quickly to upcoming new versions of languages, or even the addition of new languages to the family.

Our approach provides improvements and advantages over other possible alternatives. On the condition that all the languages considered share a common semantic foundation, we claim that:

- By grouping the languages into a family sharing a common set of constructs, we can provide an automated generic approach which can be used to build translators between any pair of those languages.
- Thanks to the use of a lightweight semantic definition technique, we can automatically obtain a simplified verification tool, that performs an initial first-pass assessment of the correctness of the translation.
- The way our technique builds and consolidates a language family, allows for intensive reuse of program transformations.

1.3 Solution

In this dissertation we combine several established software engineering techniques to address the specific problem of producing families of program translators.

We use a software architecture product-line approach to build a generic framework for the production of families of program translators. This framework can be used for different families of languages, by parameterising it with the specific language constructs.

The internal structure of the product-line allows for maximal reuse of shared language constructs. For every language that is included into the product-line we reuse the existing shared constructs first, and only then include its own specific constructs and transformations. Thanks to the use of a set of specific language-oriented metrics, we can assess at any point how the inclusion of languages, constructs or transformations into the product-line contributes to the synergy of the system.

To generate the specific language concepts structure for every family of programming languages, we use a reverse engineering model extraction technique, contributed by a grammar convergence approach. An adapted interpretation of the same technique is used to produce the language-independent concepts model for every program being translated.

Addressing the issue of language variability, whether it is at a syntactic or semantic level, requires the use of program transformation techniques. A program may have to undergo several transformations to consistently go from the particularities of one language to the particularities of another programming language. Moreover, to gain more confidence in the translation process, the translated program should be verified for equivalence against the original program. We apply a verification technique on the original and translated programs, based on a combined approach of grammar annotations, control-flow semantics and weak-bisimulation.

1.4 Contributions

The main contribution of this work is its integrated approach to build families of program translators. We combine different techniques coming from different domains, into a consistent framework. We take advantage of techniques coming from product-line engineering, program transformation, XML, process concurrency and generative meta-programming, into a single framework to build families of program translators.

More specific contributions of this dissertation can be summarised as follows.

- It shows a prototype implementation along with the validation of a product-line of program translators for the industrially-relevant case of the family of space operations languages.
- It presents a specific set of metrics, that facilitates the process of evaluating if the different actions we take to organise the system provide positive results.
- It exploits regular annotations in SDF grammar definitions, to automatically produce a preliminary definition of a language family along with a set of semi-automatic translators.
- It presents a domain-specific language to define lightweight control-flow semantics on language grammars, and to automatically generate,

from that, labelled transition systems for equivalence verification of the translated programs.

1.5 Overview of the Dissertation

The approach we follow to produce a solution to our specific problem of language translation can be reduced to three main steps: first a study of relevant technologies to acquire the required background knowledge. Second, the development of a simplified technique for the language-to-language translation problem, and finally the generation of a complete solution to produce families of program translators.

Before explaining the core of our approach, we present in Chapter 2 several preliminary concepts that we use repeatedly along our work. We give an introduction to the grammar formalisms and techniques we use, and to relevant technologies for program transformation. We explain the Operations Languages, giving an insight in the kind of languages involved in the case study.

Next, in Chapter 3 we present an overview of current technologies used for the automatic generation of translators, and program transformation in general. We also present an overview of product-lines, as well as other techniques closely related to our subject.

In Chapter 4 we develop the basis of our technique for language-to-language translation. We explain in detail our annotated grammar notation, how to generate translators and how to complete them with additional transformations when needed. We explain as well our lightweight notation for control-flow semantics, and how to use it to generate a simple equivalence verification system.

Chapter 5 builds on the previous chapter to explain how to produce product-lines of program translators. The global approach is decomposed into four important steps, each of which are explained in detail. First, we start with a description of the product-line architecture. Next we review the grammar convergence approach to extract shared language constructs. Then, we explain how to handle language variability through program transformations. Finally we put it all together into the proposed framework. As an additional support to the process of building product-lines of program translators, we present a set of metrics specifically designed to measure compatibility among constructs and languages in a language family.

Chapter 6 presents the experiment used to validate our technique. We report on the operations languages case, and use intensively the set of language metrics to show how the languages, language concepts and constructs, converge into the product-line depending on different scenarios.

Finally, in Chapter 7, we discuss the achieved results, analysing advantages and disadvantages, strengths and limitations of the proposed technique. We draw conclusions, and sketch avenues of future work.

2 Preliminaries

This chapter collects a series of terms and concepts that we use and reference repeatedly along our work. Our intention is simply to provide an introductory explanation of these terms and concepts, deep enough to put the reader in context, and to eliminate possible ambiguities.

2.1 Programming Language

For our working definition of programming languages we are influenced by Simonyi's thoughts on Intentional Programming [4, 105].

> A programming language is a set of abstractions provided to the programmer to implement an executable solution, partial or complete, to a certain problem.

Depending on the problem, a different set of abstractions will be needed to define the solution. Because many different kinds of problems exist, as well as different opinions on how to solve them, the existence of many programming languages is a natural consequence. In our case study, for example, we are concerned by the specific branch of problems related with the execution of operations for spacecrafts in space missions.

2.2 Operations Languages

In spacecraft missions, the *procedures* used to perform any of the activities during the different phases of a space mission, are written using one among the multitude of operations languages in existence. The actual operations language used by a mission centre depends, amongst others, on the specific control equipment chosen when designing the mission.

Although different operations languages may exhibit syntactic differences as well as differences in how they interact with the mission control centre and the spacecraft, they need to conform to certain standards imposed by industry, such as the ECSS-E-70-32 standard [42]. In general, Operations Languages (OL) are used to build procedural scripts that describe high-level, goal-oriented activities to be carried out by a spacecraft. These high-level activities are built in terms of more elementary activities like telecommands and telemetries. Telecommands are instructions uploaded to the spacecraft to execute an action, and telemetries are blocks of data received from the

spacecraft, as a measurement of its current state and that of its surroundings [42, 76].

It is important to realise that telecommands and telemetries are not defined by a procedure or by an OL, but are described in a separate Mission Information Base (MIB) [115]. A procedure written in an OL thus interacts with a spacecraft by sending to the mission control centre a request to execute an instruction stored in the MIB, and optionally waiting for confirmation or data. When receiving such a telecommand or telemetry request, the control centre does a preliminary check against the MIB to confirm that the instruction is well defined and that its parameters are consistent, before sending the actual instruction to the spacecraft. It will also receive the data returned by the spacecraft, and pass it back to the procedure when requested.

In addition to this direct interaction with the control centre, OLs contain language constructs common to most imperative programming languages. They provide the ability to structure the different instructions to be executed in larger procedures. For this purpose, they contain primitives to control the flow of execution within a procedure, like branching, iteration or exception handling constructs, as well as the ability to execute instructions in parallel or sequentially. They also contain primitives for variable assignment, arithmetic operations and string handling.

Figure 2.1 shows part of a test procedure written in the PLUTO [42] operations language. As illustrated by the example, PLUTO supports conditional instructions like the `if ... then ... else` (lines 11—19), Boolean comparisons like `=` and `!=` (lines 7 and 11) and string manipulation operators like the concatenation at line 16. It also provides dedicated instructions for logging (lines 6, 12, ...) as well as dedicated command and telemetry instructions to communicate with the satellite. Examples of the latter are the telemetry instruction `Value of` (line 7) and the telecommand `initiate and confirm` (line 13).

There exist quite a number of operations languages, several of which were analysed during our case study: the Spacecraft Test and Operations Language *STOL* [51, 110], the Procedure Language for Users in Test and Operations *PLUTO* [35, 42], the language underlying the Manufacturing and Operations Information System *MOIS* [95], the User Control Language *UCL* [7], the European Spacecraft Control Language *ELISA*, or the Test and Operation Procedure Environment *TOPE* [76]. In addition, for some of these languages more than one version exist and are currently in use.

Operations languages are considered as dedicated languages for the specific domain of space operations.

2.3 Domain-Specific Language

We adopt the definition provided by van Deursen et al. [120], even though the author himself warns about the vagueness of his definition due to the difficulty of correctly defining what a problem or application domain is.

```
Initiate and confirm step Switch on Gyros
    declare
        event evtTimeout
    end declare

    Log "PROCEDURE Pluto_Test_43_03 Step_1";
    Wait until ( Value of DHT30100 = ACTIVE)
        timeout 1 h 20 min
            raise event evtTimeout;

    if (monitoring status of DHT30101 != nominal) then
          log "Enabling Gyros command: PHC10117 scheduled 18h 43";
        initiate and confirm PHC10117
           with Timetag := 2008-08-02T18:43:12.000Z end with
           refer by cmdPHC10117;
          log "result is:" + confirmation status of cmdPHC10117;
    else
        log "Gyros already enabled.";
    end if;

    watchdog
        initiate and confirm step Gyro Controller Timeout
            preconditions
                wait for event evtTimeout
            end preconditions

            log "Gyro controller was not active within time.";
        end step;
    end watchdog
End step;
```

Figure 2.1: Code fragment of a test procedure in the PLUTO operations language.

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and restricted to, a particular problem domain. [120]

In this definition van Deursen points out that the key term is the focussed expressive power of a DSL. DSLs can have a variety of characteristics. They can look just like any other programming language, or have a very specific syntax. They can be a library of specialised functions inside a general-purpose language, or have a specific dedicated compiler. Most DSLs are relatively small, precisely because they provide a restricted set of abstractions only.

The family of DSLs which we are mainly interested in, in this thesis, is the family of operations languages. They are dedicated languages that are meant to *control* the different operations that need to be executed by a satellite via a mission control system. This means that they are essentially restricted to describing the flow of control in which those operations are executed.

Whereas some operations languages, like the classic STOL [51, 110], are tuned-down versions of general-purpose programming languages, of which

essentially the control-flow constructs are used, more dedicated modern OLs, like PLUTO [35, 42] or MOIS [95], contain little more than basic control-flow primitives, and specialised operations directives.

As for the case of operations languages, domain specific languages can be grouped into languages families, depending on their application domain, and sphere of activities.

2.4 Language Family

Implementing an adequate solution for a problem involves first deciding on what are the required abstractions to implement it. Next, it is necessary to choose a programming language better suited at providing these abstractions. Apart from some particular cases, often there is more than one language that can be chosen. By restricting the nature of the problems we want to solve, to those belonging to some specific application domain, we can confine ourselves to a specific and restricted group of programming languages providing the required abstractions to solve those problems. This reduced group of programming languages is what we call a *Language Family* targeted to a specific application domain, like is the case of the family of operations languages, targeted to the spacecraft operations domain. Our definition of language family is similar to the definition of software family used by Weiss et al. [129] Weiss says that defining a family is ultimately reduced to identifying and characterising its members. A key step is to come up with an analysis of what is common to all members, and what can vary among them.

> A language family, therefore, can be informally defined as the group of programming languages that provide the set of abstractions required to solve the problems in, and specific to, some application domain.

Implicit in our definition we find one constraint that for practical reasons needs to be imposed on the members of the family. This constraint holds for our case of building language translators, and can be dismissed for other situations. To include a programming language inside a family, it must be *domain-complete*: it has to consider all the required abstractions to solve all the problems in the domain. Our approach does not deal with cases where this minimal set of abstractions is not respected. It is up to the user discretion to define how to handle those incompatibilities. On the other hand, additional abstractions can be part of the language. In practice, unless we are talking about a domain-specific language, specifically designed to satisfy the needs of some application domain, programming languages offer a wide range of abstractions, providing more freedom to the programmer. This is the case for general purpose programming languages, and is also the case for those domain specific languages that have been designed based on, or inspired by, some more general purpose languages.

2.5 Language Concept and Language Construct

As seen previously in this chapter, a programming language is defined by the set of abstractions it considers. Each abstraction represents a different class of activity, or category, with distinguishable semantics from other classes of activities.

Two levels can be differentiated for every abstraction. First, the conceptual level, which is abstract and universal. This level is represented by the core semantics of the abstraction, and is common to every language implementing that abstraction. It is the intrinsic nature of the activity, that remains unchanged from language to language, regardless of its concrete implementation in a grammar.

> *Language concepts* are the core semantics of a class of activities. Language concepts are language-independent and have distinct semantics from other language concepts. [107]

Second, the instance level is concrete and particular to every language. It describes how the abstract concept is instantiated for a particular language. This level is represented by the syntax of the productions in every language grammar. It maps the abstract, conceptual idea of a class of activities to the specific and concrete grammar of a given language. In the grammar, the abstraction receives a concrete structure.

> *Language constructs* are the syntactical structures of a language that correspond to particular language concepts. Language constructs have a distinguishable semantics from other language constructs in that language. In the grammar, a language construct can range from one to several non-terminals, and the set of productions defining them. [27, 32]

As a very simple example, *addition* is a concept, regardless of how it is defined in a grammar. For a given language the corresponding construct can be defined either in prefix notation `'+' Exp Exp -> Exp`, infix notation `Exp '+' Exp -> Exp` or as a function `'Add' '(' Exp ',' Exp ')' -> Exp`.

Categorisation of Constructs Concepts bring order to the big diversity of entities we perceive and experience. Physical objects are perceived as instances of more general concepts. For example, your desk chair is an instance of the more abstract concept of "chair". Concepts stabilise our world, by organising some of the perceptible attributes of instances, thus capturing a notion of similarity [107, 74].

The terms category and instance are important when we consider that building a language family consists of classifying the instances of concrete language constructs present in individual languages, into the conceptual categories or language concepts, present in the family of languages. Such a

category in the language family serves as a link between the different instances in each language. Specific language constructs get linked to their generic language concept.

The different abstractions used by languages to define and solve problems, are the concepts. These language concepts are independent of the programming language. What programming languages provide are constructs, that are instances of the abstract concepts. These constructs can be categorised into some language concept thanks to their attributes, specially their semantics.

Special attention is needed for compound constructs, which belong to more than one category. A typical example is for instance: Exp '+' Exp -> Exp. In some languages this *Add* construct represents both the mathematical addition and the string concatenation. These grammars do not make a specific distinction between the two interpretations, and delegate the decision to the runtime environment or the typechecker system. If some language in the family uses two different constructs for this case, then two different categories will be needed in the family.

> In our work, the categories or language concepts are the non compound abstractions provided by the languages, and the instances are the language constructs provided by the grammars. Categorisation is the process of linking language constructs to language concepts, such that all constructs sharing the same semantics, belong to the same category.

The categorisation of language constructs permits ultimately to classify the languages themselves into language families. In such a way, the generic structure of our translation system points out in the direction of grammarware engineering, whose principles we try to adhere to.

2.6 Grammars and Transducers

Grammars are the main input in our approach. Programming languages are described by context-free grammars [24, 1]. Context-free grammars can be augmented by the user with annotations producing annotated grammars [50]. Annotated grammars can be used to generate regular tree grammars [59, 29] describing the abstract syntax trees of programs. Finally, tree transducers [71] can be used to transform grammars from one type to another, and to translate programs accepted by one grammar, into programs accepted by another grammar. These definitions will be necessary in Chapter 4, where we describe our approach to language translation.

2.6.1 Context-Free Grammar (CFG)

A Context-Free Grammar (CFG) defines the syntax of a language. In our approach we use CFGs to recognise the source code of programs –presented

as strings– that adhere to the grammar, and to derive the parse trees of those programs.

Formally we define a CFG as a tuple $CFG = (\Sigma, N, s, P)$ where:

- Σ is a finite alphabet of terminal symbols.
- N is a finite set of non-terminal symbols, and $\Sigma \cap N = \varnothing$.
- $s \in N$ is the start symbol.
- P is a finite set of productions of the form $n \to C$ where $n \in N$ and $C \in (N \cup \Sigma)^*$

2.6.2 Annotated Grammar (AG)

An Annotated Grammar (AG) is a CFG augmented with a set of annotations linked to its productions. This set of annotations holds additional information related to the language. Annotations can be arbitrary objects. They have no influence on the process of parsing source code, but provide the required information for other tasks like for instance producing tree grammars and abstract syntax trees.

Formally, an AG can be defined as a tuple $AG = (\Sigma, N, s, P, A, F)$ where:

- Σ, N, s and P are defined as in a standard CFG.
- A is a set of annotations.
- F is a set of mappings $P \mapsto A$, from productions to annotations.

2.6.3 Regular Tree Grammar (RTG)

While CFGs are used to recognise strings, Regular Tree Grammars (RTG) are used to recognise trees. In our approach we use CFGs to recognise valid source code, and then, thanks to a parser, produce parse trees or concrete syntax trees. From there, and to be able to use a generic set of transformation tools, we produce an abstract version of these concrete syntax trees, where no terminal symbols or syntactic keywords are included, and where the trees' structure has been simplified. These abstract syntax trees (AST) are no longer a valid derivation for the CFG used to recognise the original source code. By working with RTGs, that can be generated automatically from an AG, we fill the specification gap between CFGs and ASTs, and we provide a clean way to validate the structure of the resulting ASTs.

An RTG is a tuple $G = (\Sigma, N, s, P)$ where:

- Σ is an alphabet of tree constructors c with a fixed arity $ar(c) \geq 0$.

  The arity $ar(c)$ of a tree constructor c determines how many children it has. Constants (like for instance a variable's name) have an arity of zero.

- N is a finite set of non-terminal symbols, and $\Sigma \cap N = \varnothing$.
- $s \in N$ is the start symbol.
- P is a finite set of productions of the form $n \rightarrow c(N_1, ..., N_k)$, where $c \in \Sigma$ and $ar(c) = k$

A tree t is said to be a valid derivation for a regular tree grammar g if:

- It is constructed from the start symbol $s \in N$
- $\exists s \rightarrow t(N_1, ..., N_n) \in P$ such that for every N_i inside t, there is a production $N_i \rightarrow u(M_1, ..., M_n) \in P$, whose right hand side can replace N_i, constructing a tree u, that is a valid derivation for some $q \in N$

As an example, having the following RTG:
$G = (\Sigma, N, R, P)$
$\Sigma = \{program, if, while, exp, com\}$
$N = \{R, I, W, E, C\}$
$P = \{$
$R \rightarrow program(I)$
$R \rightarrow program(W)$
$I \rightarrow if(E, C, C)$
$W \rightarrow while(E, C)$
$E \rightarrow exp()$
$C \rightarrow com()$
$\}$

The following two trees are valid derivations for G:
program(if(exp(),com(),com()))
program(while(exp(),com()))

2.6.4 Tree Transducer (TT)

A Tree Transducer (TT) is a recursive program that traverses and transforms an input tree by generating as output another tree, a string, or more generally any object that can be produced through a tree traversal. Essentially a tree transducer consists of a set of rules that transform an input node by applying a function that depends on the node's label. Each child is recursively matched with the set of rules to complete the transformation. We use tree transducers as the standard way to specify a tree transformation.

A (top-down) Tree Transducer is a tuple $T = (Q, \Sigma, \Sigma', q_0, \Delta)$, where:

- Q is a set of function names.
- Σ and Σ' are the set of input and output symbols respectively.
- $q_0 \in Q$ is the initial function.

- Δ is a set of transduction rules of the type $q(f(x_1, ..., x_n)) \rightarrow u[q_1(x_1), ..., q_n(x_n)]$, where:
  - $f \in \Sigma$.
  - $u \in \Sigma'$.
  - $q, q_1, ..., q_n \in Q$.
  - $x_1, ..., x_n \in N$, where N is a set of input variables.

2.7 SDF Grammars

The Syntax Definition Formalism, SDF, is a formalism for the definition of grammars, which combines lexical and context-free syntax definition. It supports arbitrary context-free syntax thanks to its underlying generalized parsing algorithm, and provides several disambiguation methods to deal with ambiguous grammars. It also supports modularization and reuse of syntax definitions [126].

An important difference between SDF and (E)BNF notation is that the left and right-hand sides of the production rules are swapped. The SDF equivalent of a BNF production $X ::= A\ B\ C$ is the production $A\ B\ C \rightarrow X$. In addition, the *right*-hand side of an SDF production can be annotated with a list of attributes that characterise that production. An example of such an attribute is the constructor attribute **cons** which is used when building an abstract syntax tree (AST) from a parse tree:

$$A\ B\ C \rightarrow X\{\textbf{cons}(ConstructorName)\}$$

where $ConstructorName$ will be used as node name in the AST.

Another important feature of SDF is the possibility to annotate non-terminals in the *left*-hand side of a production with labels:

$$\textbf{label}_\textbf{a}{:}\ A\ \textbf{label}_\textbf{b}{:}\ B\ \textbf{label}_\textbf{c}{:}\ C \rightarrow X\{\textbf{cons}(ConstructorName)\}$$

This last feature is useful to avoid certain mapping problems when, for instance, matching non-terminals in source and target productions do not appear in the same order. Figure 2.2 shows an example of an annotated SDF production.

```
"if" cond:Expr
"then" true:Stats
"else" false:Stats
"end if"
    -> If {cons("IfThenElse")}
```

Figure 2.2: An example of a production in SDF.

2.8 Grammarware

The term Grammarware has been coined by Klint et al. [56] as an answer to the need for a grammar-aware software engineering.

Grammarware comprises the terms grammar and grammar-dependent software. Grammar refers to any kind of grammar formalisms and notations. Not only the grammars defining the syntax of a programming language, but in general any kind of structured format definition. Grammar-dependent software refers to any kind of software that intrinsically requires and makes use of grammar knowledge. A parser generator is a typical example.

Even though we have been using grammars in different kinds of software for many years, we still do not have a comprehensive foundation for grammarware engineering. There is a lack of best practices in general, and more specifically we do not have a discipline of programming for grammarware or a comprehensive theory for transforming and testing grammarware. There is a lack of metrics and other quality notions, and we are still looking for a unified framework relating grammar forms and notations[56]

To cope with the mentioned deficiencies, a set of common principles is proposed by Klint et al. [56]. These principles come directly from best practices used in contemporary software engineering. These best practices though, are not being used consistently in grammarware. It is therefore not a new proposal, but a call for adapting a set of common-sense principles and applying them in grammar engineering as well. These principles can be integrated into a proper grammarware life-cycle.

The proposed life-cycle starts by getting base-line grammars independent of any use case. These grammars are then customised depending on the requirements of the specific use case. The grammar-dependent software is implemented then, based on these customised grammars. Evolution in grammarware should be handled through automated transformations, such that any modification on the grammar structure is transposed directly to the grammar-dependent software components and to existent data.

Our project in general, and the tools we require and develop, more specifically, are largely shaped by grammars. We tried therefore, besides our specific scientific objectives, to adhere the higher-level goal of supporting the grammarware principles.

2.8.1 Grammar Recovery

> Grammar recovery is the structured process of deriving a language grammar from available resources like language documentation or compiler code [63, 64, 38].

This thesis does not deal with the problem of generating the base-line grammars for the languages, because it is not central to the problem of generating language translators. We do consider necessary nevertheless, to include this step into the product-line for completeness.

Our approach is fundamentally grammar-based, which is the main reason why we work with the grammarware methodology. In grammarware, the life-cycle starts by getting a base-line grammar. Grammar recovery focuses in recovering correct base-line grammars such that the grammar life-cycle is enabled. The first and main input for our entire process are language grammars we use for different purposes. Base-line grammars provide concrete and abstract grammars. Concrete grammars generate parsers for the programs. Abstract grammars provide language concepts, that constitute the core of the product-line. Grammars in general are central to our process for generating language translators.

Our approach assumes that, for every language we will work with, a correct SDF grammar is available. In practical terms though, this is not always possible. In our case study, for instance, we had to deal with the following situations:

- We did not receive, for *any* of the OLs we considered, a working grammar ready to use. The commercial nature of OLs and the sensitive nature of the space operations environment were the main reasons. Grammars needed to be extracted from the provided documentation, such as manuals.

- SDF grammars are not of common use in industry. In general, LL compliant BNF grammars were provided in the languages documentation. Extracting and transforming these provided grammars from BNF to SDF was necessary.

- Some of these BNF grammars suffered from a relatively common problem in software projects. The documentation was inconsistent and outdated with respect to the actual software. Some of the test programs we received did not parse with the grammars extracted from the documentation.

As we can see, there are cases where a grammar recovery process comes in handy. We do not go into more details in this section, but in the chapter dedicated to the product-line approach we will provide some more technical information on how we used this technique.

2.8.2 Grammar Convergence

> Grammar convergence is a lightweight verification method for establishing and maintaining the correspondence between grammar knowledge ingrained in all kinds of software artifacts. [66]

Ideally, the development process of software artifacts should follow an organised approach, like for instance the model-driven methodology: starting from a generic, abstract model, this model is transformed step by step in a controlled way, until the more specific products are generated.

In many cases, for instance, software artifacts have embedded grammar-like knowledge that was used to build them. For those cases, grammar convergence can be used to extract grammars from all these artifacts, and later on, transform these grammars until they become identical, and therefore convergent.

For the specific case of program translation developed in this thesis, we will use grammar convergence to extract the common model of the family of languages, as well as to represent in detail the differences among the different grammars, to implement the variation points.

Other complementary cases of grammar convergence exist as reference, like the introductory example presented by Lämmel and Zaytsev in [66]. In that example, different grammar formats for a small language are converged together. There is also a very complete case study that the same authors present in [67], where they analyse the consistency of the Java Language Specification among different versions and representations, using the grammar convergence technique to come up with a central model of the different grammars, that precisely and systematically represents the various differences among them.

Grammarware is complemented with a product-line approach to provide not only an organised software structure, but also an effective way to reuse the different software elements and language components.

2.9 Program Equivalence

In our specific case of program translation, an important concern is to determine if two programs, an original program and its translated version, present the same behaviour. In other words if they are equivalent.

Defining equivalence does not has a simple answer. This issue has been analysed by many authors, for instance Buss et al. [20] and Blass et al. [15] in their discussions about algorithm equivalence.

In general terms, two programs P and Q can be considered as equivalent if we can put them in an equivalence relation $\Re$, noted as $P \Re Q$. Then we can say that P and Q are $\Re$-equivalent. What we need to decide upon is which, among the existing equivalence relations at hand, should be used as $\Re$.

Different notions of equivalence exist and some are stronger than others, depending on how strict are the criteria the two programs must satisfy to be seen as equivalent under that notion.

In our study we are interested in the family of behavioural equivalences, whose basic idea, as noted by Bernardo et al. [14] is to capture whether two systems are able to mimic each other's behaviour stepwise. From the different approaches to behavioural equivalence [14, 36, 22], we are particularly interested in *observation equivalence*, also known as *weak bisimulation*.

Behavioural equivalence approaches are commonly applied on Labelled Transition Systems (LTS) [54]. An LTS is a specialised representation of the

more general notion of a Control-Flow Graph (CFG).

In our approach for program equivalence, we first build the CFGs of the programs; from there we produce the LTSs, and finally we use this LTS representation to verify program equivalence.

2.9.1 Control-Flow Graph (CFG)

We define a CFG as the tuple (V, L, A, s_0) where:

- V is a finite set of nodes or vertices v
- L is a set of labels l ($\varepsilon \in L$ is the *empty* label).
- A is a finite set of directed edges or arcs (v, l, w) between nodes. (v, l, w) represents an edge from node v to node w labelled with l. The edge (v, w) is the shorthand notation for (v, ε, w)
- $s_0 \in V$ is the initial node of the control-flow graph.

2.9.2 Labelled Transition Systems (LTS)

An LTS is a tuple $(S, A \cup \{\tau\}, \rightarrow, s_0)$ where:

- S is a countable set of states s
- A is a set of observable actions a
- τ is the *hidden* action, denoting an event internal to the system. These actions are invisible for other communicating systems or external observers.
- $\rightarrow$ is a transition relation $\rightarrow \subseteq S \times (A \cup \{\tau\}) \times S$ where $s \xrightarrow{a} s'$ denotes $(s, a, s') \in \rightarrow$
- $s_0 \in S$ is the initial state

An LTS is *non-deterministic* if $\exists (s' \neq s'') : s \xrightarrow{a} s' \wedge s \xrightarrow{a} s''$. Inversely, in a *deterministic* LTS for every state s and action a there is at most one state s' such that $s \xrightarrow{a} s'$.

2.9.3 Observation Equivalence (Weak Bisimulation)

To better understand observation equivalence, it is convenient to start defining trace equivalence and (strong) bisimulation. Trace equivalence presents the base notion of observing the behaviour of a system. It allows also to introduce the notion of hidden actions and implicitly that of external observers. Bisimulation is a stronger form of equivalence that provides a solution to the limitations of trace equivalence. Finally, weak bisimulation reconciles bisimulation and hidden actions, to provide a good compromise between strength and flexibility.

Trace Equivalence

The trace of a system S is a finite sequence of actions $a_0, ..., a_k$ such that a possible execution of S is the sequence of transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_k} s_k$. We write $traces(S)$ for the set of possible traces of S.

If for two systems P and Q it holds that $traces(P) = traces(Q)$, then they are *trace equivalent*: $P \; \Re_t \; Q$. In other words, every possible trace for P is a valid trace for Q and vice versa.

Weak Trace Equivalence

A variation of trace equivalence is weak trace equivalence, $\Re_{wt}$, where all the internal actions of the system, τ, are not considered in the trace and therefore $p_0 \xrightarrow{a} p_1 \xrightarrow{b} p_2 \; \Re_{wt} \; q_0 \xrightarrow{a} q_1 \xrightarrow{\tau} q_2 \xrightarrow{b} q_3$.

Bisimulation

Two systems P and Q can be considered bisimilar, or in a strong bisimulation equivalence $\Re_\sim$, written $P \sim Q$, if whenever one of them can execute an action, the other one can execute the same action as well.

We can say that $P \sim Q$ iff for every $p \sim q$ the following conditions hold:

- If $p \xrightarrow{a} p'$ exists, then there exists a q' such that $q \xrightarrow{a} q'$ and $p' \sim q'$
- If $q \xrightarrow{a} q'$ exists, then there exists a p' such that $p \xrightarrow{a} p'$ and $p' \sim q'$

Weak Bisimulation

As with trace equivalence, a variation of strong bisimulation is weak bisimulation, $\approx$, also known as observation equivalence, where only observable actions are matched.

Assuming we define $s \overset{a}{\Rightarrow} t$ as $s \xrightarrow{\tau}{}^* s' \xrightarrow{a} t' \xrightarrow{\tau}{}^* t$, where $\xrightarrow{\tau}{}^*$ signifies a probably empty list of successive transitions $\xrightarrow{\tau}$, then we can say that $P \approx Q$ iff for every $p \approx q$ the following conditions hold:

- If $p \overset{a}{\Rightarrow} p'$ exists, then there exists a q' such that $q \overset{a}{\Rightarrow} q'$ and $p' \approx q'$
- If $q \overset{a}{\Rightarrow} q'$ exists, then there exists a p' such that $p \overset{a}{\Rightarrow} p'$ and $p' \approx q'$

which is essentially the same definition as $P \sim Q$ but for the weaker transition relation $\Rightarrow$.

Trace Equivalence versus (Weak) Bisimulation

Trace equivalence falls short in two aspects. First, in the presence of non-determinism it can evaluate as equivalent systems that have different behaviour. For instance in Figure 2.3, $traces(P1) = traces(P2)$, even though their behaviour is not the same: $P1$ is a non-deterministic LTS, and an

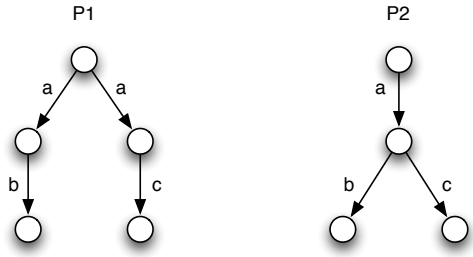


Figure 2.3: $P1 \; \Re_t \; P2$ but $P1 \not\sim P2$

external user has no choice between actions b and c. In $P2$, which is a deterministic LTS, the choice can be made by the user. Second, trace equivalence disregards the branching structure of systems because it considers only final traces.

Bisimulation solves these limitations, but it still presents one drawback regarding our needs: it does not provide enough flexibility when program constructs need to be adapted. In program translation, when we cannot establish a one-to-one equivalence between constructs in different languages, we are forced to adapt these constructs through program transformations. These transformations can introduce (or eliminate) some actions that, if included in the bisimulation, will provoke the verification to fail, even though the behaviour of the program with respect to the satellite or the control center remains the same. Weak bisimulation allows us to declare these actions that are internal to the programs as hidden actions, therefore providing an additional degree of freedom which is required in our program translation approach.

2.10 Product-line

Other than the generic language translation problem we discuss in this thesis, there is a second problem related with how, when defining translators between different languages in a family, we can take profit of the many common constructs we can find in related languages.

> A product line engineering approach implements and facilitates large-scale reuse, for improving the development efficiency of families of systems, sharing a common set of features satisfying the main needs of a specific domain. [119]

In our case, the specific domain is that of space missions operations, and the family of systems we need to develop are program translators. Since the translators act on programs designed with languages belonging to the family of operations languages, the features included in any translator belong to a

set that is largely common to all languages in the family. Reuse is thus not only possible but strongly desirable.

One of the most significant software engineering goals has been the quest for reuse. Building systems by assembling components has become a commonly accepted software development technique. Module-based approaches and later object-oriented programming have introduced reuse of rather small units of code. Large-scale reuse problems were addressed with object-oriented frameworks and component-oriented programming that showed up as parallel approaches. The notion of software product-lines resulted from combining software architecture and component-based software development.

Creating a product-line requires some essential activities following the notions of functionality-based architectural design [17]. First we have the domain engineering, or core asset development, whose goal is to establish the capability of the product-line to generate the products, in our case program translators. Three outputs are expected from this activity [49]:

1. Scoping. A description of the products that will be included into the product-line. It will consider commonalities among products, and how the products vary from one another. The scope of our product-line concerns the set of translators that will be generated. This set is restricted to the languages in the family. How and how much the translators differ, depends on specific components of each language.

2. Core assets. They include the product-line architecture, that will satisfy also the needs of the individual products, by defining the variation points to support the different products considered in the scope. Other core assets are the software components that will be systematically reused for the development of products, as well as requirement specifications and domain models. Core assets in our approach are the grammars of the languages in the family. Other assets like language concepts and constructs are derived from those grammars, and program transformations are driven by the grammar structure.

3. Production plan. This is the set of processes attached to every core asset, defining how the asset will be used for product development. The production plan describes how the individual processes have to be put together to build a product. The production plan for our program translators product-line starts with the construction of the required grammars, passing by the definition of the product-line structure, until the production of the resulting translators.

Product development, or application engineering is the second essential activity required to create a product-line. This activity takes as input the three outputs described above, plus the requirements for the individual products, which can be seen as the variation from some generic product description detailed in the scope of the product-line.

Ultimately, the product-line approach consists of correctly developing a probably big group of desired products. This is accomplished by first receiving the requirements of a product within the scope of the product-line, and then following the production plan to make a proper use of the core assets.

Our specific product-line will generate a set of program translators within the scope of the operations languages family. It will follow a grammarware oriented production plan, that will make a proper use of grammars, language concepts and constructs, and program transformations. The details of the product-line approach will be provided in Chapter 5, only after the language-to-language technique is explained in Chapter 4.

2.11 Conclusion

This chapter presents the most relevant concepts and ideas that we use along this thesis. It tries, first, to define more precisely certain notions, like languages and families, concepts and constructs, grammarware and product-lines.

Second, it presents this concepts in an order that already shows the direction followed by the thesis and its approach. Starting from the individual programming languages, we analyse their grammars and dissect them in their basic elements. Then we put those elements back together in an organised common structure, before generating and verifying the final product.

3 Related Work

Many techniques can be used to build program translators. These techniques range from string manipulation techniques, using text processors like `sed` and `awk` [37], or general-purpose programming languages like `perl` [127], over specific transformation languages like TXL [30], to more structured approaches based on some kind of program specifications, using tools like Rose [97].

Although the distinction among techniques to build program translators is not clear, we propose a classification based on the kind of language specification they are based on, and on the main transformation technique they use to manipulate the program's structure.

We explain the technique used in each category, thanks to a toy example where we build a simplified translator between two small and similar languages. First we introduce the example, and then we explain each different technique to build translators, using the example to illustrate how to build such a translator for the two languages.

3.1 Running example.

The example we are presenting follows our hypothesis that automating the process of building program translators requires a high similarity between the languages involved. Domain-specific languages inside a single domain are in general expected to share many characteristics. These similar characteristics can be exploited when building a translator.

We will use two very similar languages, where we can establish a full equivalence mapping between their constructs. The idea is to show what can be automated, and how, by using different techniques and approaches. The languages for the example are very small: there is only one "representative" construct for each language (an in-depth analysis of translation will be the subject of Chapter 6).

In this section, we want to discuss the advantages and disadvantages of different translation techniques, providing a common ground for a subsequent comparison with our own technique, presented in the following chapters. The reader should be aware that each of these techniques may provide additional, more advanced functionality, not shown in this example. We believe though, that this simple example allows the reader to get a good understanding of the techniques presented, without requiring a deep knowledge of those techniques.

We call our languages L_A and L_B. Both languages are completely equivalent, at a semantic level. The differences between them are only syntactic. Each language consist of a single conditional construct that, depending on the evaluation of a conditional statement producing a true or false boolean result, will execute one block of instructions from the two provided possibilities. In the example we are not unfolding the constructs related with the conditional statement nor with the block of instructions. We assume that the translation is already solved at those points.

For both languages we provide an EBNF syntax, as well as a compact semantics definition. We assume that the input programs have been parsed and appear as an abstract syntax tree, which is traversed depth-first left-to-right for interpretation.

Whenever this example will be used to build a translator, in the following sections of the chapter, we assume the translation is unidirectional from L_A to L_B, unless otherwise explicitly stated.

To facilitate further references, Figure 3.1 presents a compact EBNF syntax of both languages, and Figure 3.2 presents the corresponding semantics definitions.

The semantics definitions assume that: $[\![\quad]\!]$ is the semantic function describing the semantics of a program; a superscript L delimits the function to the context of a language L; a subscript f represents an auxiliary function for specific fragments of a program; capital letters represent variables holding a program or a fragment of a program. For instance, $[\![X]\!]_f^L$ applies the auxiliary function f to the program fragment X in the context of language L.

Program	::= If
If	::= "if" Expr "then" Stats "else" Stats "fi"
Expr	::= <boolean expression>
Stats	::= <sequence of instructions>

(a) L_A syntax

Procedure	::= Eval
Eval	::= "eval" "(" Cond ")" "[" Block "," Block "]"
Cond	::= <boolean expression>
Block	::= <sequence of instructions>

(b) L_B syntax

Figure 3.1: Compact L_A and L_B grammars.

$$
\begin{aligned}
[\![Program(X)]\!]^{L_A} &= [\![X]\!]^{L_A}_{program} \\
[\![If(X)]\!]^{L_A}_{program} &= [\![X]\!]^{L_A}_{if} \\
[\![(E,Y,Z)]\!]^{L_A}_{if} &= [\![Y]\!]^{L_A}_{stats} \text{ if } [\![E]\!]^{L_A}_{exp} = TRUE \\
[\![(E,Y,Z)]\!]^{L_A}_{if} &= [\![Z]\!]^{L_A}_{stats} \text{ if } [\![E]\!]^{L_A}_{exp} = FALSE \\
[\![X]\!]^{L_A}_{expr} &= TRUE|FALSE \\
[\![X]\!]^{L_A}_{stats} &= < value >
\end{aligned}
$$

(a) L_A semantics

$$
\begin{aligned}
[\![Procedure(X)]\!]^{L_B} &= [\![X]\!]^{L_B}_{procedure} \\
[\![Eval(X)]\!]^{L_B}_{procedure} &= [\![X]\!]^{L_B}_{eval} \\
[\![(C,Y,Z)]\!]^{L_B}_{eval} &= [\![Y]\!]^{L_B}_{block} \text{ if } [\![C]\!]^{L_B}_{cond} = TRUE \\
[\![(C,Y,Z)]\!]^{L_B}_{eval} &= [\![Z]\!]^{L_B}_{block} \text{ if } [\![C]\!]^{L_B}_{cond} = FALSE \\
[\![X]\!]^{L_B}_{cond} &= TRUE|FALSE \\
[\![X]\!]^{L_B}_{block} &= < value >
\end{aligned}
$$

(b) L_B semantics

Figure 3.2: Compact L_A and L_B semantics.

3.2 Translators Techniques

Developing program translators is a process that can be compared to building a compiler [3], because it can be decomposed in the following activities: scanning, parsing, semantic interpretation, transformation and code generation. Every activity in the process, regardless of the specific input and output languages of the programs to translate, contains a generic part, independent of the languages involved in the translation. These generalities between languages have been analysed over time, and specific solutions automating those parts of the process have been implemented and included in program translators. Developing translators has become, thus, more and more structured, and less ad-hoc. Nevertheless, even with those improvements, building a program translator, regardless of the technique we are using, requires an amount of problem-specific programming. In general there are differences between two programming languages that cannot be directly solved by a generic approach, and therefore a specific solution has to be designed.

This dissertation studies how to automate the process of building translators. We therefore focus our presentation on how different techniques handle what we consider amenable to automation. For each case we present pros and cons, and in Section 3.3 we use this information to discuss the advantages and disadvantages of these approaches against our own technique.

3.2.1 Ad-hoc Techniques

Even though there are cases where a pure ad-hoc approach can be the only alternative for building a translator, we are not considering them. We can always reuse existing tools and techniques to produce a translator. For instance, scanning and parsing are problems already solved. We can build our own lexical scanner and tokeniser from scratch, programming a state machine, or we can use Lex [111] and simply write the scanner description. The same applies for the syntactic analysis. On the one hand we can build a recursive descent parser on our own, or we can just write a Yacc [111] description for the language.

The real need of a so called ad-hoc approach starts with the semantic analysis. At this point, and thereafter, building a translator gets more or less ad-hoc. The differences between approaches depend on the technique used, and that is the main subject of this chapter.

3.2.2 Attribute Grammars and Compiler Compilers

This is among the most well-known and used techniques to recognise and manipulate source code. Attribute grammars were born in the mid 60's [58] as a way to define programming languages semantics. They attach attributes and semantic actions to the rules of a non-circular context-free grammar. Attributes allow to share information among the nodes in the parse tree, which can be used by the semantic actions when executed. Attribute grammars are the base for compiler-compilers tools like Yacc [111], SmaCC [19] or ANTLR [90].

Generally speaking these techniques are composed of a lexical analyser, that scans the input looking for basic text tokens. On top of it, a parser builds a parse tree based on a non-circular context-free grammar. Finally, a semantic engine executes the actions attached to the production rules, as soon as the parser provides enough information to trigger them. Thanks to the defined attributes on the non-terminals, the information resulting from the execution of the semantic actions, is passed along the parse tree until the parsing is finished and a final result is available.

In Figure 3.3 we can see an example written in Antlr defining a simplified translation between our reference languages: from L_A to L_B. The example decribes the translation from the If construct (lines 3 to 7) into the equivalent $Eval$ construct (line 11). On line 3 we define the name of the production. Line 4 declares the variable or attribute that will carry the resulting value up in the tree. Line 5 defines the body of the production, and lines 6 and 7 constitute the semantic action that will be executed. In this case the actions build the string with the translated code. The Target grammar on line 11 is not used nor necessary for the translation in this technique, and is presented here just to better illustrate the expected result. The translator is built specifically for going from L_A to L_B, and in general no verification mechanism is provided directly by these tools to assert the result. From our

```
/* Source grammar */
...
if
returns [String value]
    : 'if' e=expr 'then' s1=stats 'else' s2=stats 'fi'
    {$value = "eval(" + $e.value + ")";
     $value += "[" + $s1.value + "," + $s2.value + "]";};
...
/* Target grammar */
...
eval : 'eval' '(' cond ')' '[' block ',' block ']';
...
```

Figure 3.3: An example of an Antlr translation.

point of view, one of the main inconveniences presented by this technique, is that the semantic actions are too tightly embedded with the grammar. This makes the resulting grammar difficult to read and maintain, and also difficult to adapt for using with other languages.

3.2.3 Term Rewriting Techniques

Term rewriting techniques and systems are widely used for program transformation. As for attribute grammars, these techniques generally use BNF-like, context-free grammars to specify the syntax of the language. Thanks to this specification, programs in that language are parsed, and a complete parse tree is built before attempting any kind of manipulation or analysis on the source code. This building of a full parse tree prior to any transformation creates some memory overhead for big parse trees. But it also has interesting advantages, among which the fact that it allows for a better design and reuse by separating the specification of the syntax from the semantic actions. Once the parse tree is built, the rewriting engine will try to apply the rewriting rules, until a normal form is reached, where no more transformations can be applied. Rewriting rules are in general of the form

$$L = R \quad if \quad C_1, C_2, \ldots$$

stating that whenever the term L is matched, it can be rewritten to the term R, on the condition that $C_1, \ldots, C_n$ all evaluate to true.

Many rewriting systems exist like for instance TXL [30], the ASF+SDF Meta-Environment [118], Rascal [57] and Stratego [124]. With the exception of TXL, the other three systems use the Syntax Definition Formalism (SDF) to specify the language grammars and also share the SGLR [126] generalised parsing technique. This sharing SDF and SGLR is quite handy for it allows us to use different (though similar) rewriting techniques, with the same input.

ASF+SDF and the ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [118, 117] is a development environment for the generation of systems for constructing language definitions, and support tools for these languages. The ASF+SDF Meta-Environment allows the definition of syntactic and semantic specifications for a language, that are fully integrated together.

Syntax Definition Formalism, SDF, is a grammar formalism used by the ASF+SDF Meta-Environment, to provide the context-free syntax definition of languages. It was presented in Section 2.7.

Algebraic Specification Formalism, ASF, is a formalism for defining conditional rewrite rules. These rewrite rules can be used to define an "operational" semantics, for a language specified in SDF, through equations that can be executed as rewrite rules of the form

$$L = R \quad when \quad C_1, C_2, \ldots$$

stating that whenever L is matched, it can be rewritten to R, on the condition that $C_1, \ldots, C_n$ all evaluate to true. A simple form of equation is the unconditional one $L = R$. In the left-hand side, right-hand side and conditions of an equation, variables can be used. Matching a left-hand side of an equation implies binding the variables to the matched subterms in the concrete syntax tree. See [118] for a more detailed description.

Figure 3.4 shows a small piece of code in ASF+SDF which we will use as an example of how to build a simple translator from L_A to L_B. We have included both productions in a single file to simplify the explanation. Lines 4-5 show the "If" production on L_A and lines 7-8 show its equivalent "Eval" production on L_B. ASF+SDF is a strongly typed language, so we need to define a function for every transformation to guarantee that the translation will be type preserving. Lines 12 to 14 define these functions. To handle the different subtrees when manipulating the parse tree, while respecting typing, we have to define variables that will hold the subtrees. We do that on lines 18 to 20. Finally we need to define the rewriting rules that will perform the transformation. In this simple case, where we assume that both constructs are completely equivalent, we only need one single rule, written on lines 24 to 29. Line 24 defines the tag of the rule. Line 26 defines the pattern-matching expression in the L_A language. Notice that ASF+SDF works with concrete syntax, and therefore we write the pattern as if it were source code. Finally line 29 defines the replacement expression in the L_B language.

Some advantages of term rewriting compared against the attributed grammars approach, presented in the previous section, are: first, the term rewriting semantic actions, defined by the rewriting rules, are not tangled with the syntax, which allows for a modular design. Second, in term rewriting, the grammar of the target language is explicitly used to check for type equivalence. Some disadvantages of term rewriting are that the type checking

```
context-free syntax   %% (SDF syntax rules)

  %% Language 1
  "if" Expr "then" Stats "else" Stats "end if"
      -> If   {cons("IF")}
  %% Language 2
  "eval(" Cond ")" "[" Block "," Block "]"
      -> Eval {cons("IF")}

context-free syntax   %% (rewrite functions)

  f(If)                                          -> Eval
  f(Expr)                                        -> Cond
  f(Stats)                                       -> Block

variables

  "$Expr$"                                       -> Expr
  "$Stats1$"                                     -> Stats
  "$Stats2$"                                     -> Stats

equations

  []
  %% From Language 1
  f(if $Expr$ then $Stats1$ else $Stats2$ end if)
  =
  %% To   Language 2
  eval( f($Expr$) ) [ f($Stats1$), f($Stats2$) ]
```

Figure 3.4: An example of a simple translation function in ASF+SDF.

produces some programming overhead, and for more complex cases, it is not easy to produce intermediate results. We need to carry a lot of information in the rewriting rule conditions.

Tom / Gom

One last tool that could be interesting to mention in this section is Tom / Gom [10]. The Tom language is an extension to Java that provides it with term rewriting capabilities based on pattern matching. Tom introduces in Java constructs, basically to define data structures, to build terms, and to match and rewrite terms. Tom allows defining abstract syntax trees using Gom, and provides an adaptor to Antlr that allows Tom to build an abstract syntax tree based on an Antlr grammar, and therefore to work with terms from any language accepted by Antlr. Tom offers the possibility to be embedded in other programming languages like C or ML for instance.

Discussion

In general, term rewriting systems are an excellent choice to transform programs in the same language, because they can provide strong type checking guaranteeing structure preserving, one-to-one translations. On the other hand, this same characteristic could be problematic when translating between different languages: types can vary significantly, forcing too many additional type conversions. Being a declarative technique, it provides a straightforward way to define transformations. Nevertheless, pure rewriting makes carrying information along the tree very difficult. In general, the underlying information is represented as singly linked lists, therefore nodes have no access to their parents or previous siblings. This fact implies that querying an arbitrary position is not possible unless we keep explicit track of the traversed path for every position. Operations requiring more than only sequential processing become thus very cumbersome to define.

There are some examples of translators built with term rewriting techniques, as reported by Alalfi et al. [5]. Alalfi and colleagues do an interesting work bridging data models and UML, and build a specific tool to translate from SQL schemas to UML entity relation models. The tool is built using TXL to perform syntactic modification from a SQL DDL schema to a XMI file. The tool only supports MySQL schema and XMI version 2.1.

3.2.4 Graph Rewriting Techniques

Graph rewriting techniques can be considered as a generalisation of term rewriting techniques [101]. In the previous section, we saw that term rewriting techniques are based on tree structures, where terms can be seen as subtrees. Trees are special cases of graphs, more specifically undirected, ordered, acyclic graphs, with unlabelled edges. To exemplify, our source graph could be the syntax tree of a program accepted by some language "L_A". The graph rewriting rules or productions will be applied to this source graph, according to a defined strategy, to step by step modify the source graph until the target graph is reached. This target graph could be a syntax tree respecting the structure of some "L_B" language. Graph rewriting techniques use, very similarly to term rewriting techniques, rules of the form

$$LHS = RHS$$

stating that whenever the subgraph LHS is matched, computing a morphism against the graph that is being transformed, LHS can be replaced by an image of the subgraph in RHS. Rules in graph rewriting systems do not require to be completely defined, and only consider those parts that are relevant to the transformation. The transformations, then, basically apply the following summarised actions: nodes in RHS matching nodes in LHS are preserved; nodes and edges in LHS without a match in RHS have to be deleted; nodes and edges in RHS with no match in LHS have to be created. Several graph rewriting systems exist that share this technique,

like Progres [101] or Fujaba [43]. In Figure 3.5 we can see an example of a small graph definition, along with a rewriting rule to perform the translation, written in Progres. Lines 3 to 5 show the definition of an IF construct in a L_A language, while lines 9 to 11 show an equivalent construct in a L_B language. Lines 15 to 27 show the rewriting production that will execute the translation. A few other small examples of program-like translations performed with graph rewriting techniques can be found in [102].

```
1  /* L_A definition */
2  ...
3  node_class IF is_a STAT end;
4  edge_type has : IF [1:1] -> EXPRESSION [1:1];
5  edge_type has : IF [1:1] -> STATEMENTS [2:2];
6  ...
7  /* L_B definition */
8  ...
9  node_class EVAL is_a CFINSTRUCTION end;
10 edge_type has : EVAL [1:1] -> CONDITION [1:1];
11 edge_type has : EVAL [1:1] -> BLOCK [2:2];
12 ...
13 /* Rewriting rules */
14 ...
15 production If_to_Eval() =
16         '1 : IF   --has--> '2 : EXPRESSION
17                   --has--> '3 : STATEMENTS
18                   --has--> '4 : STATEMENTS
19   ::=
20         1' : EVAL --has--> 2' : CONDITION
21                   --has--> 3' : BLOCK
22                   --has--> 4' : BLOCK
23   redirect --has--> from '1 to 1';
24   redirect --has--> from '2 to 2';
25   redirect --has--> from '3 to 3';
26   redirect --has--> from '4 to 4';
27 end;
28 ...
```

Figure 3.5: An example of a Progres graph definition.

Graph rewriting can be seen as more powerful than term rewriting, and it indeed provides some advantages like for instance: allowing multiple disconnected subgraphs which could permit us to carry intermediate results in the same graph, avoiding complicated rules; a less rigid data structure than terms that permits incomplete node construction, again avoiding complicated rules with the possibility to use simpler rules one after the other; explicit support for local-to-global and global-to-local transformations thanks to partial-defined global-scoped patterns.

Of course, disadvantages also exist, some of them provoked by the same advantages we have just seen: incomplete pattern definitions and intermediate graphs dependency can cause ambiguities and inconsistent rewriting; to the extent of our knowledge, they do not provide a parsing mechanism for

programs, therefore besides the grammar used for parsing the program, we need additionally the graph definition.

3.2.5 Model Driven Techniques

Model driven techniques are focused on the creation and transformation of models that describe how a system is composed: its elements and relations. More general models are iteratively transformed into more specific models, until a final product like a software module or a database definition is generated. A model always conforms to a metamodel, and a metamodel can be defined in many formalisms, like the grammars used for term or graph rewriting or in more specific approaches, like the Unified Modeling Language, UML [52] or the Model Object Facility, MOF [82].

One of the standards for model transformation has been defined by the Object Management Group, OMG. It is the Query/View/Transformation, QVT [62] also known as the MOF/QVT. QVT is the standard defining an imperative and declarative language for querying and transforming MOF models. Among its implementations are Great [25] and Tefkat [68].

Model driven techniques are very similar to Grammar rewriting techniques. In many cases, they use graph rewriting systems as an implementation means, like for instance the tool Moflon [6] that uses Triple Graph Grammars [102]. Other preferred techniques for implementing model-driven transformations are functional and logic programming [77].

We can easily establish some similarities with programming languages, which is the domain that concerns us. We can say that a model is like a program, a metamodel is like the grammar of the programming language that accepts a program, and the MOF is like BNF that defines the grammars of programming languages. With this in mind, we could use Model Driven Engineering approaches, MDE, to tackle our problem, by considering program translation as a special kind of model transformation. Related work in this field can be found mainly for model transformation an model evolution.

Gray et al. [48] transform legacy C++ source code by building a specific translator from the Embedded Systems Modelling Language transformation rules to the DMS [12] Rules Specification Language. Even though this example shows a nice way to go from a model transformation to a program transformation, the approach is rather generative than translational because at the implementation level the source model transformation rules are never created and then translated. The target program transformation rules are rather generated directly with a model interpreter.

In the field of model migration De Geest et al. [33] report on a semi-automatic mapping mechanism used to support DSL evolution between versions. This project tries to be more generic and be used for arbitrary DSLs, but currently it can provide automatic mappings only for entities of the same kind, and requires user intervention for complex cases.

A similar approach focused on data model migration is used by Vermolen et

al. in [123] to map data model transformations to data migrations programs in Java. This case is focused on coupled evolutions, but is strongly related to our approach because it uses SDF grammars to define the models.

An appealing graphical approach based on colored petri nets [53] concepts is provided by Weinner et al. with TROPIC [131]. TROPIC uses the concepts of places, tokens and transitions, to define the model transformations. These concepts are derived from the elements in metamodels, models, and in the transformation logic. The presence of certain elements in the source model triggers the transition, and transports the tokens from the source, to the corresponding elements in the target model. This model driven approach gives to the transformation designer extended debugging possibilities, on the model transformations. The translation process itself, remains user driven, however.

In Figure 3.6 we can see a summarised example of a simple Tefkat model transformation, working on our running example. Lines 3 to 7 show the definition of the IF structure in the L_A metamodel, while lines 10 to 15 show the equivalent EVAL structure in the L_B metamodel. Lines 19 to 25 show the transformation rule that will execute the translation.

3.2.6 Template Based Techniques

Template based techniques are transformation techniques that act on a source tree or document, transforming it into a result tree, through the successive application of a pattern-matching / template instantiating mechanism. Several tools exist using this mechanism like XSLT [55], Velocity [109], or Andromda [112].

In this category the approaches followed by these techniques vary significantly. For instance Andromda is tightly linked with the MDE approach, while Extensible Stylesheet Language Transformations, or simply XSLT, focus on XML [40] documents, and is probably one of the better known and widely used techniques in this category.

XSLT is a declarative language for transforming XML documents into other (but not restricted to) XML documents. XSLT does not modify the source document, but generates a new result document. It uses XPATH [106] patterns to define the nodes that once matched, will serve to instantiate the transformation templates. In general template-based techniques work like this: a pattern matches one or several nodes from the source tree; then the nodes resulting from the matching are used to instantiate the template, that starts the process of creating the result tree, by directly writing pieces of it, or by instantiating other templates.

In Figure 3.7 we can see how a translation can be accomplished in XSLT, on our running example. Lines 3 to 10 show the schema definition of the L_A language, while lines 14 to 21 show the definition of the L_B language. In both cases we are defining the languages in XSD [128] which is normally used to validate an XML document. Lines 25 to 31 show the template transforming

```
/* Source metamodel */
...
<eClassifiers name="If">
        <eReferences name="exp"
            eType="#//Expression" lowerBound="1"/>
        <eReferences name="then"
            eType="#//Stats" lowerBound="1"/>
        <eReferences name="else"
            eType="#//Stats" lowerBound="1"/>
</eClassifiers>
...
/* Target metamodel */
...
<eClassifiers name="Eval">
        <eReferences name="cond"
            eType="#//Condition" lowerBound="1"/>
        <eReferences name="true"
            eType="#//Block" lowerBound="1"/>
        <eReferences name="false"
            eType="#//Block" lowerBound="1"/>
</eClassifiers>
...
/* Transformation specification */
...
RULE If2Eval
  FORALL   If if
  MAKE     Eval ev
  SET      ev.cond = if.expr,
           ev.true = if.then,
           ev.false = if.else
;
...
```

Figure 3.6: An example of a Tefkat model transformation.

this specific construct.

Regarding XML documents, in [69] Leonen builds an automated XSLT translator using the definitions of source and target structures. This approach is rather for document transformation because there is the restriction that both definitions must belong to the same class of documents. There are also other approaches more focused on program translation as shown by Clark in [26], which gets support from a knowledge base to establish the mapping. Some other efforts exist, focused on the Java programming language, that through an XML representation of the source code, as in JavaML [9], provide the possibility to translate from Java to similar languages. One such example is XES [1], that uses XSLT templates to translate from Java to the XES representation to C#. Template approaches, specially those using XSLT based techniques, are being boosted thanks to the energy dedicated by the web community on improving the related tools and on creating more tool support. In particular we believe that the XML-XPATH-XSLT com-

[1] http://sourceforge.net/projects/xes/

```
/* Source schema */
...
<xs:complexType name="If">
  <xs:sequence>
    <xs:element name="Expression" type="Expression"/>
    <xs:element name="Stats" type="Stats"/>
    <xs:element name="Stats" type="Stats"/>
  </xs:sequence>
</xs:complexType>
...
/* Target schema */
...
<xs:complexType name="Eval">
  <xs:sequence>
    <xs:element name="Condition" type="Condition"/>
    <xs:element name="Block" type="Block"/>
    <xs:element name="Block" type="Block"/>
  </xs:sequence>
</xs:complexType>
...
/* Transformation template */
...
<xsl:template match="If">
  eval( <xsl:apply-templates select="Expression"/>  )
  [ <xsl:apply-templates select="Stats[1]"/> ,
  <xsl:apply-templates select="Stats[2]"/> ]
</xsl:template>
...
```

Figure 3.7: An example of a XSLT transformation.

bination provides an interesting and powerful set of tools and techniques that can be applied to the program translation field. A drawback, nevertheless, is the verbosity of the approach, that makes documents, programs and transformers cumbersome to read.

3.3 Conclusion

Many techniques exist to build program translators, all of them offering powerful mechanisms. We can choose from traditional techniques to work with languages and programs, like attributed grammars or term rewriting, to more innovative techniques, not necessarily linked to language engineering, like the MDE, or the XML-like approaches. In every technique we can find pros and cons.

Attribute grammars technique are among the best known techniques to parse and transform programs. They provide strong parsing techniques, like Antlr and Yacc that provide some flavours of generalised parsing. One drawback of the technique is the fact that semantic actions are attached directly to the syntax definitions. This makes it harder to modularise and

reuse the grammars, and more importantly, semantic actions are directly tied to the parse tree, which in general is more verbose than the abstract syntax tree.

Term rewriting techniques, like those provided by Stratego and ASF+SDF, have strong parsing techniques as well, namely the SGLR. Its use of SDF grammars facilitates the building of modularised grammar definitions, and they keep semantic actions at a different level, not tangled with the syntax. A strong advantage is the fact of giving the possibility to add user-defined annotations to the syntax, that can be exploited later. Some of these annotations, for instance, allow very easily to directly produce an AST from the parsing. Their rewriting techniques have the advantage of being completely declarative, with recursive exhaustive transformations. Nevertheless, it is hard to freely jump to a specific point in the tree while transforming it, making it cumbersome, even with the use of improved traversal techniques, to program certain transformations.

Model and Graph techniques, regarding our needs, perform the same. They are more focused on model transformations, therefore providing interesting graphic capabilities. These techniques are in general not oriented to program source-code, but to systems architecture, typically object oriented systems. According to what has been done in previous works, the use of these techniques for program transformation requires the participation of additional program specific techniques like grammar processing tools. Certain model extraction techniques can be useful for our language families problem.

XML based techniques have many advantages that can be exploited for our work. The XML format for documents is becoming more and more accepted, even considering its extreme verbosity, that makes it definitely heavier than for instance the ATerm format used by SDF. Many tools exist to work with XML format, and even some of the classical program transformation frameworks like Stratego provide tools to convert from their internal format to XML. Template-based techniques like XSLT provide an interesting mechanism of rewriting, similar, though less powerful, to term rewriting techniques. More important, many programming languages provide support for DOM and XPath, which gives wider possibilities to build specific program transformation solutions.

Considering our special needs, we believe that we must not stick to a single technique, but rather combine several of them to exploit their advantages, and minimise their disadvantages. We need to build families of translators, which implies we should strive for reuse. We will be dealing with probably very different families of languages. This means we need strong parsing techniques. The kind of transformations required to go from one language to another can become very complex as well, and a compromise between simplicity and powerfulness has to be reached.

Regarding the first step in the translation process, parsing, we have decided to use SDF grammars for the following reasons:

- They provide a strong parsing mechanism, the generalised parsing, allowing us to define a broad range of languages.
- They accept user-defined annotations without the need of adapting the SDF core definitions, and provide a way to extract and process the annotations in a separate step independent from parsing.
- SDF comes with a simple way to define concrete and abstract syntax trees, and provides tools to convert them to different formats, depending on the user needs.
- SDF is highly modular, allowing reuse of grammar definitions.

For the next step, transformation, the choice is more complicated. We have opted for building our own library of specific transformation functions, presented to the user as a language extension to JavaScript. We work basically on a XML-DOM based representation of programs, making intensive use of XPath and E4X libraries. The main reasons for this choice are:

- JavaScript more relevant features for our approach are dynamic typing, that facilitates the translation between mapped grammar productions; first-class functions along with run-time evaluation permits to define independent components that can be put together at need for every translator; E4X complete support, and finally, easiness of linking with existing Java libraries and programs.
- Adaptability and portability. The availability of DOM-XPath support in different programming environments allows to easily port our solutions if necessary, with minimal effort and variability. It is easier for instance, thanks to the tools availability, to go from XML to ATerm and apply Stratego specific transformation techniques if needed.
- This combination present the best compromise to implement global transformations. Thanks to XPath we can easily reach any point in the program tree, without having to specifically track traversals. This is a major benefit considering the kind of transformations we are implementing.

We strongly believe that this compromise among related techniques provide us with the best alternative to build families of translators. We adopt techniques proved for program transformation, and we enrich them with complementary techniques coming from other domains.

Next chapter puts in practice what we have learned thanks to this analysis, and explains how we use the chosen techniques and tools in our approach to generate program translators.

4 Language to Language Translation

Automatically translating source code from one programming language to another while preserving the semantics is hard [114]. Only a few of such translation tools exist for a very limited set of source and target languages. Still, there are cases where such tools are needed and where constructing them is feasible. In this chapter we describe our approach to generate automatic translators between two languages inside a language family. Since we assume that languages inside a family share a common semantic basis, the translation problem is mostly syntactic in nature. This enables a high degree of automation: the translators can (largely) be generated from annotated syntax definitions. The techniques described in this chapter are the basis for our larger goal of generating families of translators, explained in the following chapter.

4.1 A Grammarware Approach

Our approach relies on ideas and principles of grammarware development [56], to automate the generation of program translators for a family of languages.

We provide a solution for two general issues we have detected, that are ultimately related with reusability. First, we want to efficiently detect and reuse the common elements between languages. Second, we want to adapt the incompatible elements thanks to a simplified and reusable mechanism. This process is constantly driven by an analysis of the principles and rules underlying the grammars used to define the languages in a family of languages.

The approach reconciles three related problems. One is the generic language translation problem [65, 113, 114]. A second problem is how to assess the correctness of the generated translators, by verifying the equivalence of the translated programs. Thirdly, when defining translators between different languages in a family, there should be a way to take profit of the many commonalities among related languages.

This chapter focuses on the first two problems: language to language translation and equivalence verification. We leave the third problem, that is based on the solution for the first two problems, to be discussed in the next chapter.

To solve the first problem, we automate the process of building program translators by taking advantage of language similarities. We map source to

target languages by annotating their grammars, and provide these annotated grammars to our system to produce an automatic translator.

To tackle the second problem, we augment our annotated grammars model with a set of generic lightweight semantic annotations, that we use to verify the translation. More specifically, we annotate operations language[1] constructs with their control-flow semantics. Next, we test the control-flow equivalence of original and translated programs, by comparing their respective control-flow graphs. We generate these graphs automatically from the aforementioned semantic annotations and use weak bisimulation [108] to show their equivalence.

In the next chapter, we will then explain how to increase sharing of language components between translators. We design a common structure for a family of languages, allowing us to classify both common language constructs and reusable program transformations. This common structure provides a generic semantic model common to our family of languages. As such, we can map directly common constructs, and we can share program transformations between languages.

For a better understanding of the different building blocks of our approach we start this chapter by presenting an overview of our basic approach to build a single translator between two languages. We introduce this way the basic idea of generating translators from annotated grammars, upon which our extended approach for languages families, explained in the next chapter, is based. In the remainder of this chapter, we then present the annotated grammars technique, which is the pillar of our approach, in more detail.

4.1.1 The APPAREIL Approach

The flow of our APPAREIL [2] process for producing a language translator, as depicted in Figure 4.1, can be decomposed in five main steps.

First (1) we start from the language documentation that provides the specifications and definitions of the languages. Since often this documentation is not complete or too informal to be used directly, we need to complete it and correctly structure it. This first step corresponds to the recovery of a base-line grammar, as recommend by the grammarware principles. This step is included in the process for completeness. In our approach we assume that the working grammars exist. If it is not the case, a grammar recovery process should be implemented, as explained in sections 2.8.1 and 5.2.1.

Next (2), we customise the grammars by using the ASF+SDF Meta- Environment [118] as a support tool to design a working SDF grammar for both source and target languages. We annotate those grammars with extra annotations defining the important language constructs and their control-flow semantics. This step is explained in Section 4.2.

[1] This is the family of languages which we studied in detail in this thesis

[2] APPAREIL stands for "Approche paramétrique de réingénierie logicielle."

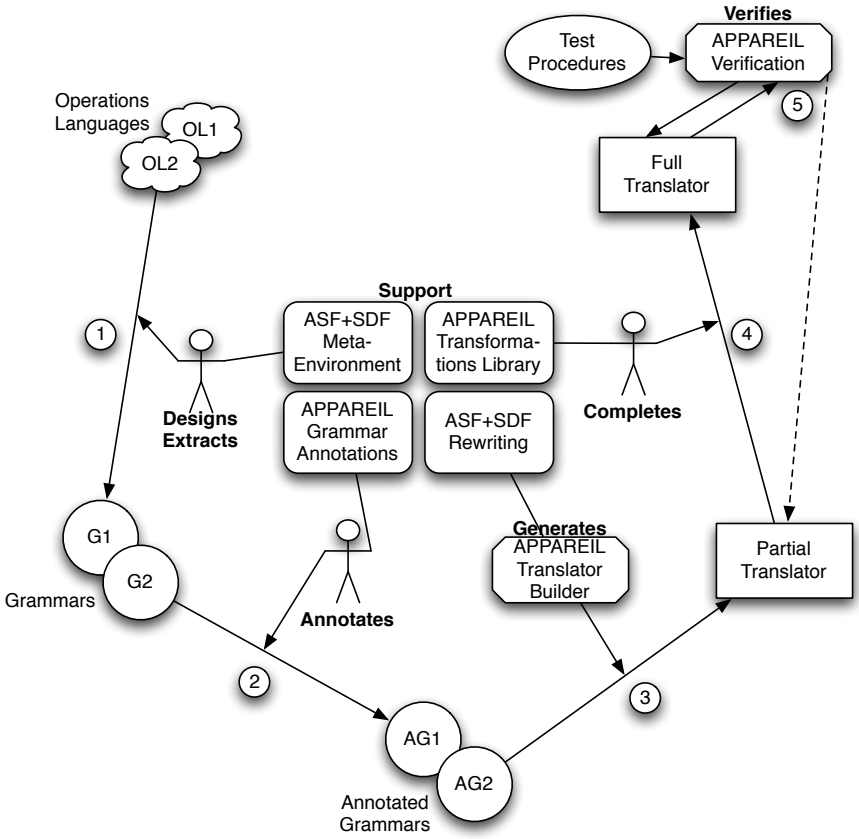


Figure 4.1: An overview of the APPAREIL language-to-language translation approach.

The third step (3) takes as input the annotated grammars of both languages, and produces a partial translator from one language to the other. This translator, depending on how different both languages are, could require a certain amount of manual extra work to complete it with additional transformations. Section 4.3 explains this step.

In step (4) we cope with those cases where the nature of the mismatches between languages are such that an automatic transformation cannot be derived only based on annotations. The previous step already provides relevant information, extracted from an analysis of the mappings, signaling the places where mismatches and incompatibilities have been found. We build the additional transformations, from components in a dedicated transformations library, to include in the translator. Section 4.3.2 exemplifies this part of the process.

Finally (5), we use the now complete translator, providing it with the test

program procedures to translate. We verify the result of this translation with our verification module, that will check for (observation) equivalence between original and translated procedures. Section 4.4 gives details on this step.

Steps four (4) and five (5) could be iterative, especially in cases when some further modifications to the translator need to be made due to errors reported by the verification module.

Figure 4.2 provides an alternative overview of our automated approach for generating language translators, now focusing on the three different kinds of actors involved. The bottom or base level represents the end users. The

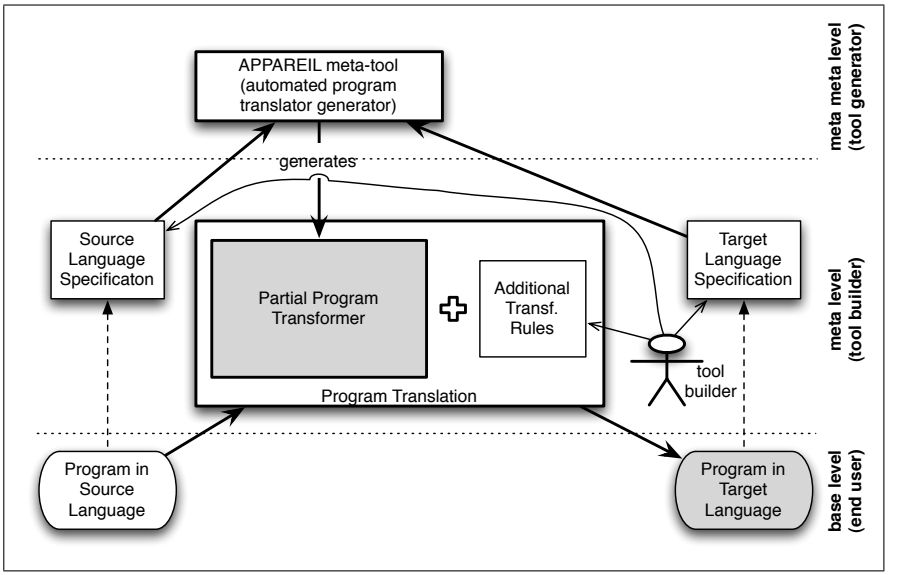


Figure 4.2: Automated generation of program translators: A schematic overview.

end users' only concern is having a translator which receives programs in a source programming language and which produces an equivalent program in a target language. At the intermediate or meta level, we have the program translator builders who provide the end users with a transformation tool for the source and target programming language of their choice. To build such a program transformation tool, program translator builders in turn make use of our *APPAREIL* meta tool which is situated at the top (or meta meta) level. The program translator builders provide this meta tool with a specification of the grammar of both source and target language, augmented with annotations that specify the correspondence between language constructs in both languages. Using this input, the meta tool semi-automatically generates a dedicated transformer for translating programs from the source to the target language, and the translator builder has to intervene only to specify

how to translate those cases for which no direct equivalence could be stated between constructs in the source and target grammars.

Although many existing tools could be used to implement the translator generator part of our solution, such as DMS [12], TXL [30] or Stratego [124], we chose the ASF+SDF Meta-Environment for implementing the generation of program translators. ASF+SDF, as explained earlier, is a specification formalism composed of the Algebraic Specification Formalism (ASF) and the Syntax Definition Formalism (SDF), allowing the integrated definition of syntax and semantics of a programming language [118] in a modular way. The modularity of ASF+SDF enables reusability, at the syntactic as well as at the semantic level, which is one of the advantages of using it as our implementation medium. Furthermore, ASF+SDF has a strong notion of syntax-directed translation both on input and output sides. Both sides are based on grammars which ensures the syntactic correctness of the results.

The following section comes back to step 2, depicted in Figure 4.1, and explains how the grammars of source and target languages are augmented with annotations defining the important language constructs, to automate the translation process.

4.2 Annotated Grammars

Syntax-directed translation [2] is a common mechanism used, mainly in compiler construction, to translate from a source to a target language. A particular instantiation of this technique is the use of syntax-directed transduction [70], which specifies the input-output relation of the translation and deduces the actual translator from that relation.

The Step 3 (from Figure 4.1) of our approach builds on these techniques to develop a simple and easy-to-use mechanism to semi-automatically build source-code translators between two related languages. We take as input context-free grammars of both languages, previously annotated with constructor and label information, to establish a mapping [91] between corresponding language constructs (step 2). The mechanism automatically generates the translator for most of the syntactic constructs (step 3), and provides support to manually extend that translator for the others (step 4).

Our technique assumes that the languages between which we want to translate are semantically similar. Furthermore, the more similar the syntax of source and target language (i.e., the language grammars), the less human intervention will be required to produce the complete program transformer (step 4).

4.2.1 Grammar Annotations

As explained in Section 4.1.1, to automate the translation process of programs between two operations languages, we will use SDF grammars and annotations to define the languages, and ASF rewrite rules to specify the

translator. For example, the language constructs shown on lines 4 and 6 of Figure 3.4 on page 47 belong to two different operations languages. Although the syntactic structure of both constructs differs, they have the same semantics: they evaluate a boolean expression, and depending on its truth value, they execute one of the statement blocks. Both constructs belong to one single abstraction or language concept. Therefore, we declare a correspondence between them by annotating both productions with the same constructor name *cons("IF")*.

As Figure 4.3 illustrates, such an equivalence can be regarded as an abstract syntax tree (AST) shared by the corresponding constructs in both languages. Since terminals (denoted by octagons in the figure) do not interest us when defining this correspondence, they are left out of the common AST. In this way we build a mapping between the two languages, allowing us to translate specific instances of a construct in one language to its counterpart in the other language.

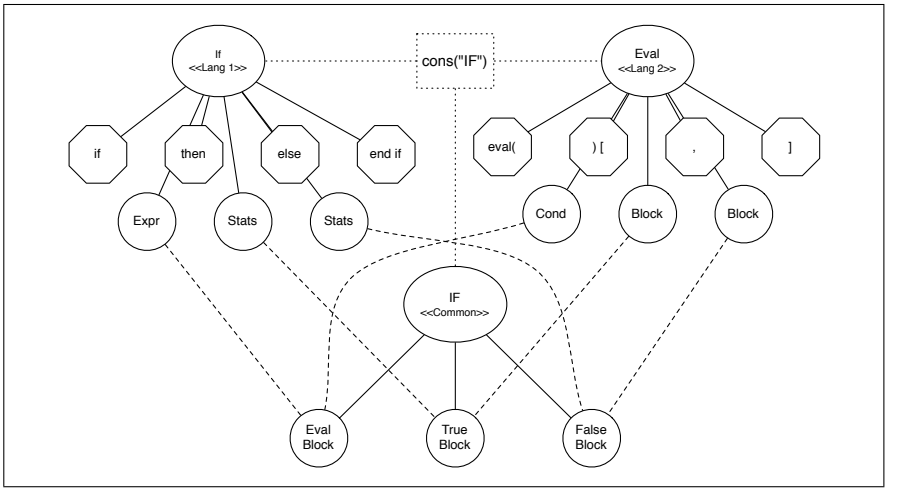


Figure 4.3: Common Abstract Syntax Tree.

There are equivalences in the left-hand sides of these SDF productions as well, like is the case with the conditional expression (`Expr` in language 1 and `Cond` in language 2), and with the blocks of statements (`Stats` in language 1 and `Block` in language 2). In most cases, the system can guess automatically how to match non-terminals occurring in the left-hand side of both productions. The order in which they appear is often enough to establish a one-to-one mapping. Only in more complicated cases, for example when there are multiple non-terminals of the same type that appear in a different order, we need to manually label the corresponding non-terminals explicitly (as was illustrated in Figure 2.2 on page 31).

Some Annotated Examples The basic idea of annotating the grammars is represented in the small example from figures 4.4 (textual) and 4.5 (graphical). We have two equivalent constructs, *SWait* and *TWait*, in two different *Source* and *Target* grammars. The square shapes in the graphical representation, represent keywords, that are not relevant for this example. The ellipses represent the Non-Terminal Symbols that contain the representative information we need to translate, and the diamonds represent the annotations used to link both constructs together. The full figure represents two *Wait* constructs whose semantics are to hold the execution of the program until the *cond* expression becomes true, which allows the program to continue with the execution of the next instruction. *timout* represents a timeout control value that once elapsed, continues the execution, even if *cond* never becomes true.

```
"waituntil" "("  cond:SExp ")" ""  timout:STimeout ""
    -> SWait cons(" Wait")

"<Wait>" "<Until>"  cond:TExp "</Until>"  timout:TTimeout "</Wait>"
    -> TWait cons(" Wait")
```

Figure 4.4: Linked grammars (textual representation).

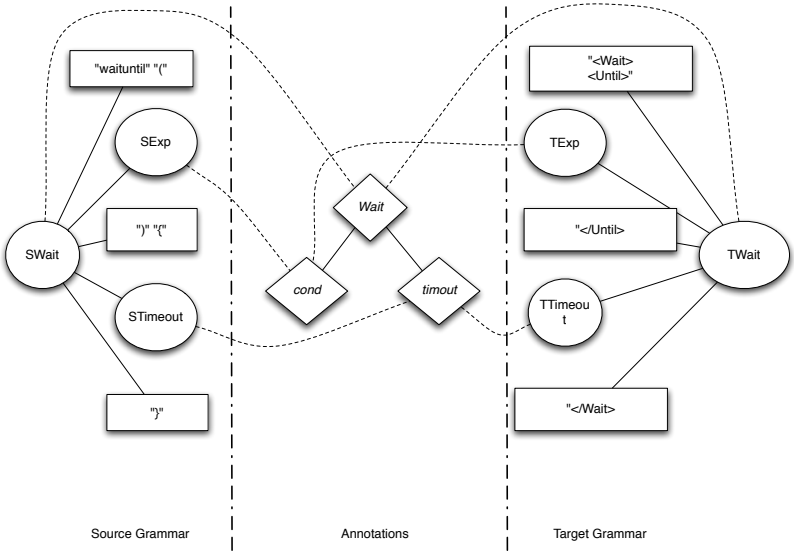


Figure 4.5: Linked grammars (graphical representation).

The annotations representing the link between these two constructs are in our approach basically unrestricted, with the exception of certain reserved

keywords, or SDF's own lexical restrictions. We can use whatever word we consider useful to annotate the constructs at the constructor level as well as at the label level.

To better understand how the annotations must be included, and how they represent an abstraction or language concept, let us analyse a well known *If-Then-Else* conditional construct, present in almost every language:

```
"if" Expression "then" Block "else" Block "fi" -> If
```

The language concept underlying this conditional construct, can be structurally represented according to Figure 4.6.

```
<If>
  <condition/>
  <trueblock/>
  <falseblock/>
</If>
```

Figure 4.6: A Language Concept structure.

This representation (Figure 4.6) implies that regardless of syntax specific characteristics, and considering the minimal conditions required to execute the *If* construct, it is composed of a conditional expression *condition*, and of two groups of instructions *trueblock* and *falseblock*. If the result of evaluating the *condition* expression yields *true*, then what is inside *trueblock* will be executed, otherwise, what is inside *falseblock* will be executed. In this approach, the *If* construct, or concept, does not care about what is really inside its internal components. It is up to *If's* internal components to evaluate the information inside them as necessary. Reduced to the extreme, the *If* component is aware only of the order of execution of its internal components, and knows only how to perform a simple conditional test.

To build the language concept from the grammar construct, we use the annotations. In our *If* example, the corresponding annotations are shown in bold:

```
"if"  condition:Expression
"then"  trueblock:Block
"else"  falseblock:Block "fi"
    -> If {cons("If")}
```

It is rather straightforward in this example to see how the annotations correspond exactly with the elements in the language concept in Figure 4.6. There are other examples, though, where the relation between a construct and a concept is less clear. Let us slightly modify the If example, to see how the relations can change:

```
"if" Expression Alternatives "fi" -> If
THEN ELSE -> Alternatives
```

```
"then" Statement* -> THEN
"else" Statement* -> ELSE
```

In this variation we have introduced additional non-terminal symbols: *Alternative, THEN* and *ELSE*. The *If* structure has changed such that it is no longer that straightforward to annotate the construct, according to our needs. How to define the two required blocks *trueblock* and *falseblock* now that we have only one symbol *Alternatives*? Of course in this case the easiest and recommended solution is to modify the grammars to simplify them and align them. If for some reason this is not possible for other cases, we can *not* annotate certain symbols, having the following result:

```
"if"  condition:Expression Alternatives -> If {cons(" If")}
THEN ELSE -> Alternatives
"then"  then:Statement* -> THEN
"else"  else:Statement* -> ELSE
```

Figure 4.7: An alternative *If* construct.

This combination of annotations will produce the language concept

```
<If>
  <condition/>
  <then/>
  <else/>
</If>
```

which compared with the concept represented in Figure 4.6 remains structurally the same. As you can see, though, there are name differences in the internal elements: *trueblock* becomes *then*, and *falseblock* becomes *else*. These differences were introduced on purpose, and in this case they are not important, but they should be avoided. The concepts produced by the grammar can be considered as a draft or template that will be later on personalised by the translator designer. What is important to take into account at this point are two things. First, the *If* concept will be created, therefore all the languages that will implement a construct with this semantics, should annotate the construct with the same *If* name to allow the mapping. Second, the productions *Alternatives*, *THEN* and *ELSE* will not produce concepts, because of its lack of constructor annotations. They will be absorbed by the declaring production.

4.3 Automated Generation of Program Translators

In Section 4.1.1 we already provided a schematic overview of how the APPAREIL meta-tool generates program translators. In this section we explain

step by step how to use that tool to produce a translator.

1. Make the annotated syntax definitions of source and target languages that will be given as input to the APPAREIL meta-tool.

   a) For every production in the source language, find an equivalent production in the target language, if available, and annotate both productions with the same annotation in the `cons()` section.

   b) Link the left-hand sides of both productions. For every non-terminal in the source production, label the corresponding non-terminal in the target production with the same attribute name. Note that this is only needed when non-terminals cannot be matched by order of appearance, which is the default behavior.

   c) When possible keep your grammars simple. If a better alignment can be done by modifying some productions, for instance removing unnecessary symbols, do it.

2. Run the APPAREIL meta-tool with the obtained language specifications as input.

   a) Feed the system with the annotated grammars of source and target language. Based on this information the APPAREIL meta-tool will then build an ASF+SDF translator from source to target (see section 4.3.1 for more detailed information).

   b) In addition, the meta-tool produces warnings about, for instance, unmapped productions.

3. Manual intervention: adding additional transformation rules.

   In the last step, we need to manually handle cases where mappings could not be derived automatically. We do so by manually adding rewrite equations to the translator, as will be explained in more detail in Section 4.3.2. The warnings generated by the meta-tool are useful here.

4.3.1 Transformation Example

We now illustrate the approach by deriving a translator for the toy languages shown in Figures 4.8 and 4.9. In those figures, we already performed step 1 of our approach. Both grammars have been annotated by the user with constructor information and labels.

When given these grammars as input, the APPAREIL meta-tool (step 2) starts by relating productions in the source and target grammars that have the same constructor attribute. The non-terminal at the right-hand side of the production in the source grammar becomes the argument of a translation function `f`, while the right-hand side of the production in the target grammar becomes the result of that translation function. For example, for

```
module Source
context-free syntax
"proc" b:StatsS "endproc"            -> StartS {cons("Start")}
"if" e:Expr "then" b:StatsS "fi"     -> IfS {cons("IfThen")}
"if" Expr "then" StatsS
    "else" StatsS "fi"               -> IfS {cons("IfThenE")}
"while" e:Expr "do" b:StatsS "od"    -> WhileS {cons("While")}
if:IfS | w:WhileS | e:Expr           -> StatS {cons("Stm")}
it:StatS*                            -> StatsS {cons("Block")}
"true" | "false" | "nil"             -> Expr {cons("Expr")}
"not" Expr                           -> Expr {cons("Expr")}
```

Figure 4.8: A part of the Source grammar.

```
module Target
context-free syntax
"start(" b:BlockT ")"                -> StartT {cons("Start")}
"eval(" e:Expr "," b:BlockT ")"      -> EvalT {cons("IfThen")}
"loop(" e:Expr "," b:BlockT ")"      -> LoopT {cons("While")}
if:EvalT | w:LoopT | e:Expr          -> InstT {cons("Stm")}
it:InstT*                            -> BlockT {cons("Block")}
"true" | "false" | "nil"             -> Expr {cons("Expr")}
"not" Expr                           -> Expr {cons("Expr")}
```

Figure 4.9: A part of the Target grammar.

the productions with constructor attribute `cons("IfThen")`, a translation function `f(IfS) -> EvalT` is derived.

Next, the rewrite equations for the transformation system are generated based on the left-hand sides of both productions. For the translation function `f(IfS) -> EvalT` we thus obtain the following rewrite equation:

$$\texttt{f(if \$Expr\$ then \$StatsS\$ fi)} = \texttt{eval(\$Expr\$, f(\$StatsS\$))}$$

where every non-terminal `NT` has been replaced by a variable `$NT$`. For every non-terminal, the corresponding translation function is invoked recursively.

The signatures of the translation functions (Figure 4.10) and the corresponding rewrite equations (Figure 4.11) are derived automatically from the grammars in Figures 4.8 and 4.9.

```
f(StartS)       -> StartT
f(IfS)          -> EvalT
f(WhileS)       -> LoopT
f(StatsS)       -> BlockT
```

Figure 4.10: Signatures of different translation functions.

```
      f(proc $StatsS$ endproc) = start( f($StatsS$) )
   f(if $Expr$ then $StatsS$ fi) = eval( $Expr$ , f($StatsS$) )
f(while $Expr$ do $StatsS$ od) = loop( $Expr$ , f($StatsS$) )
            f($IfS$ $StatS*$) = f($IfS$) f($StatS*$)
         f($WhileS$ $StatS*$) = f($WhileS$) f($StatS*$)
           f($Expr$ $StatS*$) = $Expr$ f($StatS*$)
```

Figure 4.11: Rewrite equations for translation functions.

4.3.2 Handling Mismatches: Manual Intervention

Finally, we illustrate step 3 of the approach. We handle all cases for which the meta-tool failed to establish a mapping between productions. For each missing construct in the source grammar an extra rewrite equation needs to be added to the automatically derived translator. For example, the production with constructor attribute `"IfThenE"` in Figure 4.8 has no equivalent production in the target grammar of Figure 4.9. Manual intervention is needed to tell the translator how to handle this language construct. A possible solution for this particular example is:

1. Modify the translation function for `IfS` by changing the cardinality of the resulting type: `f(IfS) -> EvalT+`
2. Add an equation to rewrite the pattern:
   `f(if $Expr$ then $StatsS$ else $StatsS2$ fi) =`
   `eval( $Expr$ , f($StatsS$) )`
   `eval( not $Expr$ , f($StatsS2$) )`

After this manual intervention we obtain a complete translator that can translate any program in the source language to the target language. For instance, the program in the left column of Figure 4.12 gets translated to the one on the right.

Input program in source language	*Translated program in target language*
`proc`	`start(`
`if true then nil else nil fi`	`eval(true, nil)` `eval(not true, nil)`
`while true do nil od`	`loop(true, nil)`
`endproc`	`)`

Figure 4.12: A translation example.

4.3.3 Translation Semantics

Finally, the process of first building a translator, and then using it to translate source code, is formalised in this section.

Domains

CFG = Context-Free Grammars, defined in Section 2.6.1 on page 28

AG = Annotated Grammars, defined in Section 2.6.2 on page 29

RTG = Regular Tree Grammars, defined in Section 2.6.3 on page 29

SC = Source Code, is the set of strings representing programs that can be derived by some CFG grammar. We note SC_g the set of source code strings $sc \in SC$ that adhere to grammar $g \in CFG$.

CST = Concrete Syntax Trees, is the set of trees representing programs that are a valid derivation of some annotated grammar in AG. We note CST_{ag} the set of trees $cst \in CST$ derived from a context-free grammar $ag \in AG$.

AST = Abstract Syntax Trees, is the set of trees representing programs that are a valid derivation of some regular tree grammar in RTG. We note AST_{rtg} the set of trees $ast \in AST$ derived from a regular tree grammar $rtg \in RTG$.

Operations

To produce a translator we have as main input two context-free grammars: one for the source language and the other for the target language of the translation. Before building the translator we need to produce annotated and regular tree grammars for each context-free grammar. We first present the two functions required to produce these additional grammars, and then we show how to produce the translator.

- $annotate : CFG \rightarrow AG$

  $annotate(g) = ag$, is a manual function where the user annotates a context-free grammar by hand, as previously described in Section 4.2.

- $build\_rtg : AG \rightarrow RTG$

  The $build\_rtg$ function builds a regular tree grammar from an annotated grammar. The purpose of this function is to abstract the annotated grammar into a grammar describing the abstract syntax trees. The set N of non-terminals, as well as the start symbol s are are passed directly from the AG to the RTG. The set of annotations A in the AG becomes the alphabet of tree constructors of the RTG. The main task of the function is to build the set of productions P' of the tree grammar. This is done thanks to the application of a tree transducer (see Section 2.6.4 on page 30) $rtgp$ to every production in the annotated grammar. $rtgp$ transforms the structure of each annotated production and gets rid of terminal symbols.

  $build\_rtg(\Sigma, N, s, P, A, F) = (A, N, s, P')$, where:

$P' = \{rtgp(p) \mid p \in P\}$

Foreach $p = (n \rightarrow x_1, ..., x_n) \in P$:

If $(p \mapsto a) \in F$, then

$rtgp(p) = n \rightarrow a(y_1, ..., y_k)$ where $y_i, ..., y_k$ is the largest subsequence of $x_1, ..., x_n$ where every $y_i \in N$

else $rtgp(p)$ is undefined.

A program-to-program translator is represented by the function *translate*. To obtain the *translate* function we use several auxiliary functions that process the input source and target grammars. We start by defining the translate function, and define the auxiliary functions as needed.

- $translate_{s,t} : SC_s \rightarrow SC_t$

  $translate_{s,t}(sc_s) = sc'_t$, translates a program sc_s adhering to grammar s into a program sc'_t adhering to grammar t. A translator function is the composition of four auxiliary functions:

  $translate_{s,t} =$
  $unparser_{rtg_s,ag_t} \circ trw_{rtg_s,rtg_t} \circ cst2ast_{ag_s,rtg_s} \circ parser_{ag_s}$

- $parser_{ag} : SC_{ag} \rightarrow CST_{ag}$

  A parser is a function specific to a grammar $ag \in AG$. It will process only source code $sc \in SC_{ag}$, producing concrete syntax trees $cst \in CST_{ag}$. Parsers are built by the auxiliary function $build\_parser : AG \rightarrow (SC_{ag} \rightarrow CST_{ag})$

  $parser_{ag}(sc) = cst_{ag}$ where $sc \in SC_{ag}$

  $build\_parser(ag) = parser_{ag}$

  The *build_parser* function uses an external tool to build a parser. This external tool will depend on the specific kind of grammars used to specify the languages, which in our approach are SDF grammars (See Section 2.7 on page 31). SDF grammars are processed with the set of tools related to SGLR [118], to produce a specific parser: $parser_{ag}$, for the grammar $ag \in AG$ given as input.

- $cst2ast_{ag,rtg} : CST_{ag} \rightarrow AST_{rtg}$

  This tree transducer transforms a concrete syntax tree $cst \in CST_{ag}$ into an abstract syntax tree $ast \in AST_{rtg}$, where ag and rtg are related by the function $build\_rtg(ag) = rtg$. $cst2ast_{ag,rtg}$ requires the auxiliary function $ann : CST_{ag} \rightarrow A$, where A is the set of annotations in AG.

  $cst2ast_{ag,rtg}(cst_{ag}) = ast_{rtg}$, is defined as (for brevity we drop the suffixes of the function in the definition below):

if cst is a tree $x(x_1, ..., x_n)$, then

$cst2ast(cst) =$

$$\begin{cases} \text{the tree } ann(x)(cst2ast(x_1), ..., cst2ast(x_n)) & \text{if } x \in N_{ag} \\ \text{the empty tree} & \text{if } x \in \Sigma_{ag} \\ \text{the leaf node } x() & \text{otherwise}^* \end{cases}$$

(* x is a constant or lexical token. e.g. a variable's name)

$ann(n) = a$

For a given $ag \in AG$, for every node n inside $cst \in CST_{ag}$, if n is a derivation of production p, and p is annotated with annotation a, then $ann(n) = a$, and we say that n is decorated with a.

- $trw_{s,t} : AST_s \rightarrow AST_t$

  A term rewriting system is a function specific to a pair of grammars $s, t \in RTG$, and therefore it will process only abstract syntax trees $ast \in AST_s$, producing abstract syntax trees $ast' \in AST_t$. trw is built by the auxiliary function $build\_trw : RTG \times RTG \rightarrow (AST \rightarrow AST)$.

  $trw_{s,t}(ast_s) = ast'_t$, where:

  the ast received as parameter adheres to the regular tree grammar s, and the resulting ast' adheres to the regular tree grammar t. ast is assumed to be functionally equivalent to ast': $ast \approx ast'$.

  $build\_trw(s, t) = trw_{s,t}$

  This is a manual function where the user builds by hand a term rewriting system, that adapts the constructs in a source language s, that are not compatible with a target language t.

  Having $s, t \in RTG$, for every construct in the language s that has no compatible construct in t , the user must include a transformation rule into the term rewriting system that will be produced as output.

- $unparser_{rtg,ag} : AST_{rtg} \rightarrow SC_{ag}$

  The function $unparser$ is a tree transducer that receives as parameter an abstract syntax tree ast_{rtg} that is a valid derivation for the tree grammar rtg, and produces as output a string $[a, b, c, ...]$, representing a source code sc_{ag} that adheres to an annotated grammar ag. The two grammars rtg and ag are bound by the relation $build\_rtg(ag) = rtg$

  $unparser_{rtg,ag}(ast_{rtg}) = sc_{ag}$, is defined as (for brevity we abbreviate $unparser_{rtg,ag}$ as unp in the definition below):

$$unp(ast) = \begin{cases} [x] \\ \qquad \text{if } ast \text{ is the leaf node } x() \\ \\ [\alpha^*{}_0, unp(x_1), \alpha^*{}_1, ..., \alpha^*{}_{n-1}, unp(x_n), \alpha^*{}_n] \\ \qquad \text{if } ast \text{ is a tree of the form } x(x_1, ..., x_n) \end{cases}$$

1. For the base case, when the ast node is a lexical token $x()$ (e.g. the name of a variable), *unparser* produces directly the value of the token into the resulting string.
2. When the ast node is a subtree, the produced string needs to combine the information in the node with information that comes from the production from which the node was originally derived.

   To combine this information, the following relationships related with the *ast* being processed hold:

   - The root node x of the *ast* (its constructor, more specifically) corresponds to an annotation in the set A in ag.
   - There is a mapping $p \mapsto x \in F$, where F is the set of mappings from productions P to annotations A in ag.
   - The production p in this mapping is equal to $y \rightarrow \alpha^*{}_0, y_1, \alpha^*{}_1, ..., \alpha^*{}_{n-1}, y_n, \alpha^*{}_n$, where each α^* represents a possibly empty list of consecutive terminal symbols, and where by construction each $y_i \in N_{ag}$ corresponds to one of the subtrees x_i in the *ast*
   - The symbols $\alpha^*{}_0, ..., \alpha^*{}_n$ in the set of terminals Σ in ag, will be included in the output string without further processing.

With this set of functions we can produce a translator for a source and target languages inside a given language family, provided that for every production either a direct mapping was established, or a transformation function to handle the mismatch was provided. The semantic definitions provided do not consider cases where these two conditions do not hold and partial translators need to be generated.

4.4 Control Flow Semantics

Program translation assumes some form of equivalence between the original and the translated program. Ideally, we would hope to have a full equivalence between both versions of a program. Realistically, considering that two different languages rarely offer exactly the same abstractions and that for those incompatible constructs a –probably imperfect– transformation must be implemented, our hopes of equivalence must be tuned down somewhat.

Program translation is a process where you need to transform the original structure of a program. For simple cases, the required transformation is mainly syntactical, and you just need to change keywords or symbols. For

these simple cases, as shown in Section 4.2, it is the designer of the translator who provides, as a postulate, the equivalence of two constructs by mapping their structures on a one-to-one basis.

For more complicated situations, the transformations can affect the semantics and the structure of the constructs, for instance if these constructs are not present in the target language: if a language does not natively support a construct that provides specific semantics, it may need to be implemented through complex combinations of other language constructs to simulate the required behaviour. These adaptive transformations, and their overall impact on the program, cannot always be assumed to be correct by construction, like we did with the previous case.

In the worst case, the semantic *impedance mismatch* may be so important that it is not possible to implement a proper translation at all.

Because of these problems, the translation process is often not fully automatic and requires some manual intervention to deal with the more complex cases. In that case, a way of verifying whether the transformations are correct is desired. This boils down to checking whether the original and the translated program are functionally equivalent or not.

Considering though, that full functional equivalence verification of programs is an undecidable problem [104], we need to weaken the requirements, and introduce a simplified form of equivalence.

Among the many authors that have tried to solve the full equivalence verification problem by adopting a simplified equivalence testing, we mention in the following paragraphs those that provide inspiration to our own approach.

Podlovchenko [94] reduces the equivalence problem by modeling programs as graphs specifying the succession relation between the instructions of the program. Next he defines a set of logging instructions that he maps against certain relevant nodes in the graph. Then he traverses the graph registering the history of abstract execution of the different instructions mapped with each node. Finally, his equivalence verification between two programs is established by comparing the history of the traversal of the two graphs modeling the programs.

Mason et al. [73] show another simplification of operational equivalence, treating programs as black-boxes, observing only the produced results regardless of how they were produced. Mason and his colleagues then extend the operational equivalence of programs to statements and expressions, using a constrained equivalence based on contextual assertions.

Similarly, Pitts [92] adopts a structural approach largely based on syntax. Pitts focuses on contextual equivalence of expressions, defining two phrases of a programming language as contextually equivalent if any occurrences of the first phrase in a complete program can be replaced by the second phrase without affecting the observable results of executing the program.

Finally Veerman, in his work on Cobol programs transformation [121], check the correctness of the implemented transformations by drawing the control flow graphs for the two sides of some specific rewriting rules, and

comparing the two graphs to see if they are bisimilar.

Our own simplification to the equivalence verification problem is the result of adapting the ideas form the authors we have mentioned, to our specific domain of Operations Languages. Operations Languages are used to describe high-level goal-oriented activities. To describe these activities, the procedures designed with these OLs make use of elementary instructions like telecommands and telemetries. Telecommands are uploaded to the spacecraft to execute an action, and telemetries are measurements received from the spacecraft's sensors. These elementary instructions are specific and dependent on the spacecraft, and not on the operations language used to design the procedure. What is specific to the language is how it controls the flow of execution of these instructions.

If we translate a procedure from some operations language to another, the control-flow instructions specific to the language may change depending on what control-flow primitives are available. The elementary instructions, on the other hand, must always be the same and they must respect the same order of execution both in the source and in the target language.

This dependency on the control-flow instructions provided by each language made us focus on how different operations languages manage control-flow, and made us adopt a control-flow semantics to verify the equivalence of translated procedures.

More specifically, since the semantics of operation languages is essentially a control-flow semantics, the approach we follow is to create a Labelled Transition System (LTS) –see Section 2.9.2 on page 35– for both the original and the translated program, based on the control-flow specifications of each language. Then we pass these two LTSs to the CADP Bisimulator tool [13], and ask it to check for equivalence, more specifically weak bisimilarity [108], of the two labelled transition systems. Finally, in case the bisimulation cannot be established, we analyse the trace produced by the CADP Bisimulator and we present to the user the graphs and the source code, where markers show the problems. In other words, for our domain of operations languages we reduce the problem of functional equivalence verification to the simpler problem of verifying observation equivalence (weak bisimulation) of the control-flow graphs of two programs.

To create the required Control-Flow Graph (CFG) , and later on its corresponding LTS, independently of the specific language used to write the programs, we use a small domain-specific language that can be attached to every grammar as annotations. This DSL, that we have named the Control-Flow Semantics Language (CFSL), provides the required primitives to build a CFG representing the control-flow semantics of programs written in imperative languages like, for instance, those in our family of OLs.

Our technique to generate the control-flow graphs of programs, even though conceived with OLs in mind, can be thought as generic to any imperative language rather than specific to operations languages. It provides a set of generic instructions that allow to define the arcs and edges of a control-flow

graph, based on the nodes of the abstract syntax tree of a program. Our approach to generate these control-flow graphs is similar to that of other frameworks, like for instance DeFacto [11], that uses a set of rules attached to a grammar to extract from a program a set of facts related with its control-flow structure. These facts can be used later on to generate a control-flow graph. JastAdd [39] (specific to Java) and DMS [12] are other examples of frameworks capable of generating control-flow graphs from grammar specifications.

4.4.1 Introductory Example.

Before presenting in detail the different parts of our DSL, Figure 4.13 gives a small, intuitive example of how to define the control-flow semantics of a simple program.

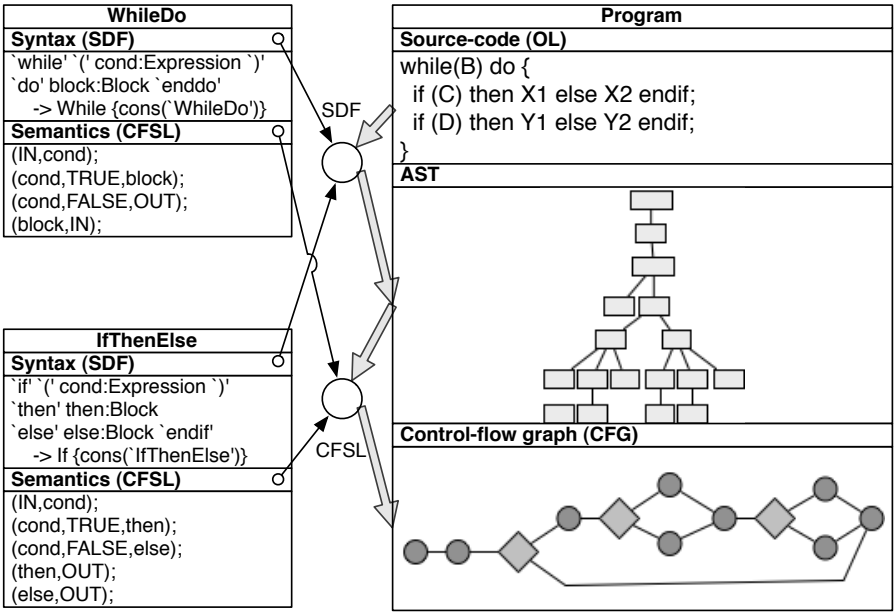


Figure 4.13: An introductory example to the CFSL.

We can see to the left of the figure two language components: a *WhileDo* component defining a loop construct, and an *IfThenElse* component defining a conditional construct. We use the term *language component* to refer to an extended description of a language construct, comprising besides its syntax, also a semantics definition. In this representation of the language components we have clearly divided the syntactic SDF definition from the more semantic CFSL definition, at the top and the bottom of each component respectively.

To the right of the figure we see the small program of our example in a box

with three different representations. At the top of the program box we have the source-code representation in some operations language. The source-code is parsed using the SDF syntactic definitions in the components, producing an AST of the program. This AST, which is the representation in the middle of the program box, is processed using the components's CFSL semantic definitions, producing the CFG representation of the program, shown at the bottom of the box.

Parsing the source-code of the program to obtain an abstract syntax tree is done thanks to SDF grammars and the SGLR parser. Since this part of the process is not central to our work, we do not elaborate on that.

This section is focused rather on how, from the AST of a program, we obtain its CFG that can be afterwards represented as an LTS. The process is sketched in Figure 4.14.

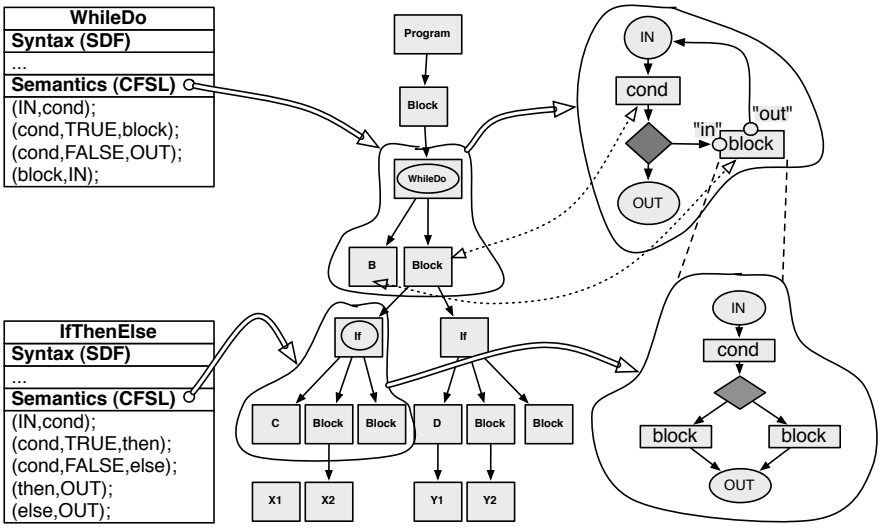


Figure 4.14: An introductory example to the CFSL.

First we perform a preorder traversal of the AST. For each node we reach during the traversal, we look into its associated language component for the corresponding CFSL definition, as shown for the WhileDo node. Thanks to the CFSL definition, we build the CFG *subgraph* associated with that node. Each child of the AST node we are processing also has a subgraph associated with it. This is shown by the dotted lines joining the `cond` and the `block` parts of the WhileDo's AST and CFG representations. Each subgraph will be in turn expanded when its corresponding AST node will be traversed, as we can see with `block`, inside the processed WhileDo node. Each of the If children inside `block` are processed thanks to their CFSL definition and the resulting subgraphs are included inside the previously generated `block` node.

Let us focus now on the CFSL instructions at the bottom of the WhileDo

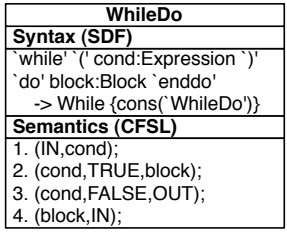


Figure 4.15: A WhileDo component.

component in Figure 4.15. The four numbered instructions (1 to 4) define each an edge between two nodes. These instructions were defined by the user manually, when designing the annotated grammar for the language. The nodes, and the edges between nodes, will be generated thanks to these instructions, taking as input the subtree which was linked with the component when the AST was created. The keywords `IN`, in instructions 1 and 4, and `OUT`, in instruction 3, define the initial and final nodes of the subgraph. `IN` and `OUT` represent the entry and exit points of the subgraphs generated by every component, and are the links with the rest of the CFG. The keywords `TRUE` and `FALSE` define the labels of the edge. Instructions with three elements, as in instructions 2 and 3, can be seen as guarded transitions: the path can be followed only if the condition (`TRUE` or `FALSE`) holds.

The intuition is that for every node reached during the traversal of an AST, we proceed as follows:

- For every child of the current AST's node, crate a new subgraph.
- For every subgraph, define "IN" and "OUT" as its corresponding entry and exit nodes.
- For every CFSL instruction, create and edge between the two subgraphs referenced by the instruction.
- For every guarded instruction (a three elements instructions), label the corresponding edge with the element in the middle of the instruction.

Graphically, the control-flow of the WhileDo example can be expressed as in Figure 4.16. This rather intuitive graphical notation will be used to better explain the control-flow graph definitions in the components, and it will be explained in Section 4.4.2.

4.4.2 Graphical Notation.

Prior to the detailed explanation of the Control-Flow Semantics Language, CFSL, this section presents the graphical notation we will use to ease the

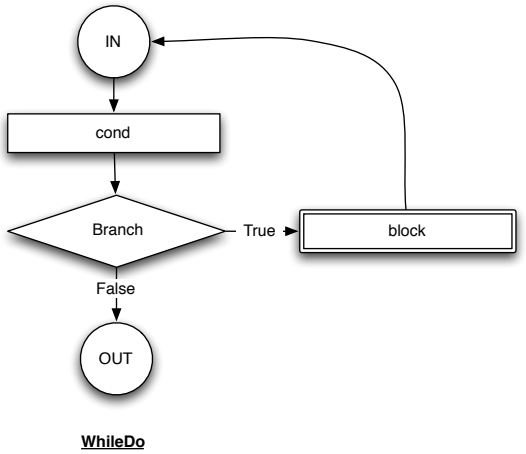


Figure 4.16: The graphical notation for the control-flow semantics of the WhileDo component.

explanation of the different constructs and instructions that constitute the languages we work with. We use this graphical notation to provide a more intuitive understanding of the control-flow inside the different instructions and their interactions. The graphic elements we use are shown on Figure 4.17.

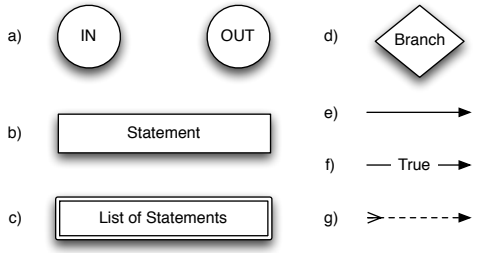


Figure 4.17: The elements in the Control Flow Graph Semantics notation.

a) The circled connectors `IN` and `OUT` delimit a construct, and pass or receive the control flow to or from another construct. `IN` receives the flow coming from the previously executed instruction, and `OUT` sends the flow to the following instruction.

   For instance, a list of instructions: `print "a"; print "b"; print "c";` will be represented as in Figure 4.18.

b) The box *Statement* designs any language element that will execute an instruction either directly, like a literal, or indirectly through its inner statements.

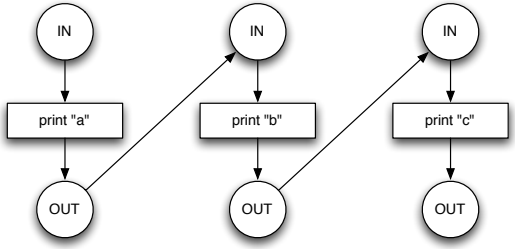


Figure 4.18: A sequence of print instructions.

A typical example of a statement is an expression, like for instance: A + B, where the top expression is an addition, and the two inner expressions are identifiers, as shown in the following abstract syntax tree representation:

```
   +
 /   \
A     B
```

While this representation is easy to understand, it doesn't say anything, at least explicitly, about the order of execution of the different parts. Defining the '+' instruction, in control-flow notation is presented in Figure 4.19, where part (a) shows the *Addition* construct. For this specific example we suppose that the two expressions that are executed first, are Identifiers, represented by the construct in part (b). Once the two expressions are evaluated, and their return values retrieved, the *Add* primitive function executes the mathematical addition.

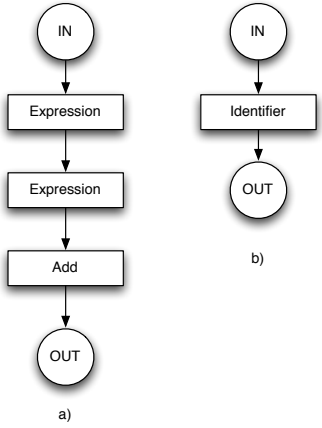


Figure 4.19: An addition expression.

For those cases where the statements in our languages are expressions or primitive functions processing values, they will not be drawn in detail. In our specific case of operations languages, we give more importance to those statements affecting control-flow, like an *IF-Then-Else* construct.

c) A *List of Statements*, represented by the double-lined box, denotes a particular kind of statement, composed of other statements that will be executed sequentially but independently, as in Figure 4.20. No statement inside the list requires the execution or the presence of other statements from the list, from a control-flow point of view. List of Statements are indeed any block, like the *then* or *else* parts inside an *IfThenEse* statement. Even if they can be modeled as a single symbol *Block*, we know that inside this symbol we have a list of statements that will be executed sequentially. The use of this symbol is not strictly necessary, but it can provide more exactitude to some definitions.

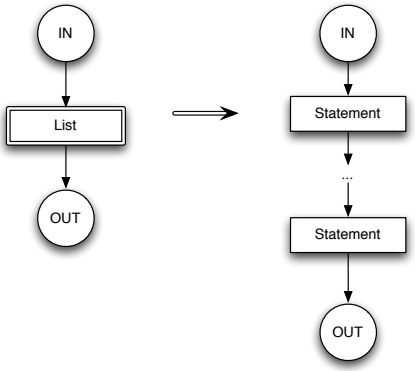


Figure 4.20: A list of statements.

d) The diamond *Branch* element conditionally sends the flow of control to one of the two possible paths *True* or *False*. This element depends on the existence of a boolean value provided by the execution of an expression immediately before the Branch. A typical use of this element is when defining an *If-Then-Else* component, as depicted in Figure 4.21.

e) The single line arrow represents the direction of a normal flow of control. It shows the order of execution of the elements.

f) The labelled single arrow represents a conditional path. In general it is used together with the diamond element (e), which will direct the control flow through the corresponding arrow, depending on its label.

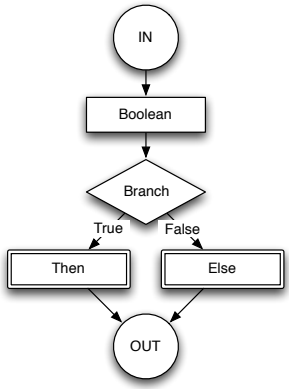


Figure 4.21: Using the Branch element.

g) The dotted line arrow is in general used with the double-line box (List of Statements) and denotes the possibility that some statement inside the list breaks the normal sequential flow, and exits the list. It is used to model *break* and *continue* statements.

4.4.3 The Appareil Control-Flow Semantics Language, CFSL.

Recalling our introductory example from Section 4.4.1, our system is composed of language components where we group the syntax and semantics of each construct present in the grammar of a language. Figure 4.22 shows the language component for the well known IfThenElse construct.

IfThenElse
Syntax (SDF)
`if' `(' cond:Expression `)' `then' then:Block `else' else:Block `endif' -> If {cons(`IfThenElse')}
Semantics (CFSL)
(IN,cond); (cond,TRUE,then); (cond,FALSE,else); (then,OUT); (else,OUT);

Figure 4.22: An IfThenElse language component.

The component in the figure is formed of two sections. At the top of the figure we have the syntax section, which shows the annotated grammar of the construct. At the bottom of the figure there is the semantics section, where we find the CFSL instructions for the component.

Regarding the syntax section, what is important to notice here are the annotations used to label the symbols in the grammar, like `cond` for the

Expression symbol, or `then` and `else` for the two Block symbols. These annotations are references that can be used in the semantics section to point to the corresponding subtrees in the Abstract Syntax Tree (AST) that hold the information to build the Control-Flow Graph (CFG).

In this part of the thesis we will focus on the semantics section of the language components, where we have the user-defined CFSL instructions. The CFSL instructions are tuples defining a transition of the LTS. The parameters in those tuples are either labels defined in the syntax section of the component, or special functions or keywords specific to the CFSL, that will be described later in this section.

We initiate the definition of our language by presenting the SDF syntax of CFSL. Remember that SDF has a functional style where the left-hand side implies the right-hand side, which is the inverse of BNF. Next we present the semantic domains and the required equations defining the semantics of CFSL.

4.4.3.1 CFSL Syntax

```
CFSLinstruction*                    -> CFSLprogram

Transition*                         -> CFSLinstruction

ComplexTransition                   -> Transition
DirectTransition                    -> Transition
GuardedTransition                   -> Transition

"(" Instruction ")"                 -> ComplexTransition
"(" State "," State ")"             -> DirectTransition
"(" State "," Guard "," State ")"   -> GuardedTransition

"TRUE" | "FALSE"                    -> Guard
"IN" | "OUT"                        -> State
AbstractSyntaxElement               -> State {1}

"default()"                         -> Instruction
"skip()"                            -> Instruction
"trace()"                           -> Instruction
"label(" Marker ")"                 -> Instruction
"jump(" Marker ")"                  -> Instruction
"ctxjump(" Marker ")"               -> Instruction
"ctxlabel(" Marker ")"              -> Instruction
String                              -> Marker {2}


{1} An AbstractSyntaxElement can be any annotation defined
in the <Syntax> section of the component.
```

```
{2} A Marker can be any String not defined as an existing
AbstractSyntaxElement.
```

4.4.3.2 CFSL Semantics

In this section we describe more formally how, thanks to the Control-Flow Semantics Language attached to a Regular Tree Grammar (see Section 2.6.3 on page 29), we can build the Control-Flow Graph, CFG (see Section 2.9.1 on page 35), for an Abstract Syntax Tree representing a program.

Domains

CFG = Control-Flow Graph. A Control-Flow Graph, $CFG = (V, L, A, s_0)$ see Section 2.9.1 on page 35), defines how the *control*, during the execution of a program, is passed between *nodes* of an abstract syntax tree, AST.

CFRTG = Control-Flow RTG. A CFRTG grammar is an RTG grammar whose productions have been annotated with a set of control-flow instructions. A CFRTG grammar holds, in its annotations, a CFSL program defining how to build a control-flow graph for every AST that is a valid derivation of that grammar.

A CFRTG is a tuple (Σ, N, s, P, F), where:

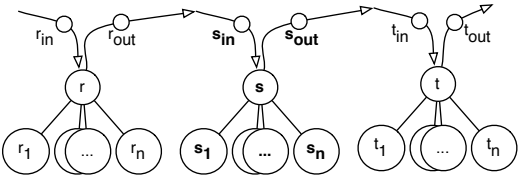
- Σ, N, s and P are defined as in a standard RTG (see Section 2.6.3 on page 29)
- F is a set of mappings $P \mapsto CFSL$, from productions to CFSL instructions (the CFSL domain is defined later on in this section).

AST = Abstract Syntax Tree. A program is represented by its Abstract Syntax Tree, AST. Every node in an AST denotes a program instruction.

Every node s of an AST can itself be considered as a tree with root node s and subnodes $s_1, \ldots, s_n$ (which can in turn be considered as trees). We use the notation $s(s_1, \ldots, s_n)$ to denote this node s with its subtrees $s_1, \ldots, s_n$. We will refer to $s_1, \ldots, s_n$ as the *internal nodes* of s. A node without internal nodes (a leaf in the tree) will be denoted as $s()$.

To define the CFSL semantics of a full AST program, we will first define the semantics of every node, in terms of the flow of control between its internal nodes. After that, the semantics of each individual node needs to be connected to that of other nodes. To achieve that, for every node we introduce two artificial internal *in* and *out* nodes. They represent the start and end points of the control flow inside the node. For a given node $s(s_1, ..., s_n)$, we thus implicitly have $s(s_{in}, s_1, ..., s_n, s_{out})$. We will use these internal in and out nodes to connect the semantics

of one node to that of another, simply by connecting the out node of one node to the in node of another, as illustrated by the picture below, which sequentially connects the control-flow of nodes r, s and t:



There is a one-to-one relationship between the nodes s in an AST derived from an RTG, and the productions p in that RTG. Since this relationship applies also to their internal nodes, and since we have introduced artificial in and out nodes, we also need to create artificial IN and OUT symbols corresponding to these in and out nodes. In other words:

$\forall s(s_{in}, s_1, ..., s_n, s_{out}) \in AST,$
$\quad \exists p(IN, p_1, ..., p_n, OUT) \in RTG$ corresponding to s
$\quad\quad$ such that s_k corresponds to p_k, for $k \in \{1, ..., n\}$
$\quad\quad$ and we assume that s_{in} corresponds to IN
$\quad\quad$ and s_{out} corresponds to OUT

How an AST node organises its control-flow internally, is described thanks to a set of CFSL instructions, that is attached to the grammar for which the AST is a valid derivation. For every node in such an AST there is a production in the respective grammar. For every production there is a set containing zero or more CFSL instructions.

DAST = Decorated AST. A decorated AST is a tuple $DAST = (AST, FD, CD)$, where:

- AST is an abstract syntax tree.
- FD is a set of mappings $s \mapsto (v, l, w)$ of nodes in the AST to arcs between nodes in that AST.

  The idea is that, for each given node s the mapping FD defines the control flow for that node s in terms of its internal nodes. The set FD is built from the CFSL instructions included in the RTG that derived the AST, as will be explained later. By building the set FD we build the arcs that will form the CFG. In other words, once some DAST has been constructed completely, the codomain of FD contains all the arcs for a given CFG.
- A set CD of mappings $s \mapsto (m \mapsto s')$, where s, s' are nodes of the AST, s is the parent of s', and $m \in Markers$ [3].

[3] according to the CFSL Syntax

Without going into the details now, this set CD is needed to provide anchors for contextual jumps, as defined later on in this section by the instructions *ctxlabel* and *ctxjump*.

CFSL = CFSL Instructions. An instruction in CFSL defines the flow of control between internal symbols of the production $p \in RTG$ where the instruction is defined. For instance, if we look at the example in Figure 4.22, the second CFSL instruction $(cond, TRUE, then)$ declares that the control will flow from the *cond* expression to the *then* block if the condition is $TRUE$.

In general, a CFSL instruction is a tuple with one, two or three elements. In a tuple (p_s, l, p_t) with three elements, defined inside some production p, p_s and p_t are symbols internal to p, and l is a label that can be $true$, $false$ or ε (empty). (p_s, l, p_t) declares that after having processed the internal flow of p_s, control will pass to p_t. This transition of flow will happen only if the value of label l was produced as the result of processing the flow of p_s.

Tuples (p_s, p_t) with two elements are simply a shorthand notation for the tuple (p_s, ε, p_t) with an empty label. We assume, therefore, that every tuple with two elements is implicitly transformed into its corresponding tuple with three elements.

Finally, tuples (c) with one element, represent complex instructions like default(), skip(), and so on. To define the semantics of such complex instructions, they will be expanded into tuples with three elements representing simple instructions [4], as will be shown further in this section.

Operations

A control-flow graph CFG is built by the function *build_cfg* from the arcs defined in a decorated abstract syntax tree. We first define the *build_cfg* function, and the *decorate* function that builds the DAST. The required auxiliary functions will be defined as needed.

- $build\_cfg : DAST \rightarrow CFG$

  The *build_cfg* function takes as input a decorated abstract syntax tree $dast = (ast, FD, CD) \in DAST$, extracts the arcs from the set FD, and consolidates them into the CFG:

  $build\_cfg(dast) =$
  $\quad initial(r) \oplus a_1 \oplus \ldots \oplus a_i \oplus \ldots \oplus a_n$
  where r is the root node of ast
  and $\{a_1, ..., a_n\} = \{(v, l, w) \mid s \mapsto (v, l, w) \in FD\}$

[4] Complex instructions are represented in the CFSL Syntax as productions for the symbol "Instruction"

The auxiliary function $initial : Vertex \rightarrow CFG$ creates, from a given AST node r, a CFG containing only that node as starting node:

$initial(r) = (\{r\}, \emptyset, \emptyset, r)$

The auxiliary function $\oplus : CFG \times Arcs \rightarrow CFG$, adds an arc (v, l, w) to a CFG:

$(V, L, A, s_0) \oplus (v, l, w) = (V \cup \{v, w\}, L \cup \{l\}, A \cup \{(v, l, w)\}, s_0)$

- $decorate : AST \rightarrow DAST$

  To build a DAST from an abstract syntax tree, the *decorate* function essentially needs to construct the set FD of mappings $s \mapsto (v, l, w)$ of nodes in the AST to arcs between nodes, and the set CD of mappings $s \mapsto (m \mapsto s')$ for context jumps.
  To build the set FD, using some auxiliary functions, we collect all CFSL instructions and convert them to arcs between AST nodes. Note that complex CFSL instructions will first need to be expanded into more simpler ones.
  To build the set CD, we need to add, for each CFSL instruction of the form $ctxlabel(m, s')$, a mapping $s \mapsto (m \mapsto s')$ from the node s having that instruction, to a mapping from the contextual label m to the internal node s' , where s is a parent node of s'. This label m can be reached by any node t descendant of s (using a $ctxjump(m)$ instruction), which is the reason why we call it *contextual* label: it can only be reached in the context of the AST node where it was declared.

  Formally, for a given ast, we have :

  $decorate(ast) = (ast, FD, CD)$

  where FD and CD are defined as follows:

  $$FD = \bigcup_{s \text{ node of } ast \wedge a \in cfsl(derived(s))} expand(s, subst(s, a))$$

  $$CD = \{s \mapsto (m \mapsto s') \mid s, s' \text{ nodes of } ast \wedge parent(s') = s \wedge ctxlabel(m, s') \in cfsl(derived(s))\}$$

  The required auxiliary functions *parent, derived, cfsl, subst* and *expand* are defined below.

- $parent : AST \rightarrow AST$

  $$parent(s') = \begin{cases} s & \text{if } s(s_{in}, s_1, ..., s_n, s_{out}) \\ & \wedge\, s' \in \{s_{in}, s_1, ..., s_n, s_{out}\} \\ \varepsilon & \text{otherwise } (s' \text{ is the root of the tree}) \end{cases}$$

  The *parent* function provides the parent node s for a node s' internal to s. If s' is the root of the AST, nothing will be provided.

- $derived : AST \rightarrow P$

The *derived* function relates nodes in an AST with the production in a CFRTG from which that node was derived.

$derived(s) = p$, where $p \in cfrtg$ is the production that was used to derive s.

- $cfsl : P \rightarrow \mathcal{P}(CFSL)$

  The $cfsl$ function provides the set of CFSL instructions attached to a production of the CFRTG grammar.

  $cfsl(p) = \{a \mid p \mapsto a \in F_{cfrtg}\}$

- $subst : AST \times CFSL \rightarrow CFSL$

  CFSL instructions are defined in terms of the productions in the grammar. To add these instructions to the DAST, we first need to redefine each instruction in terms of the AST node that was derived from the production holding the instruction.

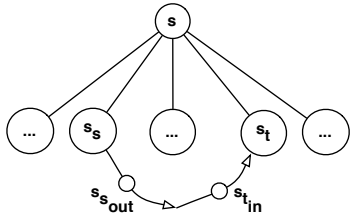
  Since $derived(s) = p$, we know that:
  $s(s_{in}, s_1, ..., s_n, s_{out}) \in AST$,
  $p(IN, p_1, ..., p_n, OUT) \in RTG$ corresponding to s,
  s_k corresponds to p_k, for $k \in \{1, ..., n\}$ and
  s_{in} corresponds to IN and s_{out} corresponds to OUT.

  If $a = (p_s, l, p_t) \in cfsl(derived(s))$, we define:

  $subst(s, a) = (h(p_s), l, h(p_t))$

  where $h(p_i) = \begin{cases} s_{in} & \text{if } p_i = IN \\ s_{out} & \text{if } p_i = OUT \\ s_{tin} & \text{if } p_i = p_t \text{ (last element of the tuple)} \\ s_{sout} & \text{if } p_i = p_s \text{ (first element of the tuple)} \end{cases}$

  Schematically, if we have a node s with a corresponding CFSL instruction (p_s, l, p_t), what we need to do is to link the inner node s_s of s to the inner node s_t of s (with label l). This is done by connecting the *out* node of s_s to the *in* node of s_t:



- $expand : AST \times CFSL \rightarrow CFSL$

  expand complements the previously defined *subst* function. *expand* only works over *complex* instructions, leaving the *simple* instructions unchanged. For each complex instruction, *expand* will call an auxiliary function with the same name as the instruction.

$$expand(s,(a)) = \begin{cases} default(s) & \text{if } a = default() \\ skip(s) & \text{if } a = skip() \\ trace(s) & \text{if } a = trace() \\ label(s,m) & \text{if } a = label(m) \\ jump(s,m) & \text{if } a = jump(m) \\ ctxlabel(s,m,s') & \text{if } a = ctxlabel(m,s') \\ ctxjump(s,m,s) & \text{if } a = ctxjump(m) \\ \{a\} & \text{otherwise} \end{cases}$$

- $default(s(s_{in}, s_1, ..., s_n, s_{out})) =$

  $\{(s_{in}, s_1), (s_n, s_{out})\} \cup \{(s_i, s_j) \mid 1 \leq i < n \wedge j = i + 1\}$

  The *default* function creates a left-to-right traversal of the internal nodes of s. It is generally used with lists of instructions that, at design time, have an unknown size. If for some production p, $cfsl(p) = \varepsilon$, then *default()* is assumed.

- $skip(s) = \{(s_{in}, s_{out})\}$

  skip provokes that the s subtree will not be included in the CFG. *skip* is generally used with *Comments* that have no effect on the control-flow of a program.

- $trace(s) = \{(s_{in}, trace_s), (trace_s, s_{out})\}$

  trace is a special instruction required when doing program equivalence verification (see Section 4.5 on page 91). *trace* inserts in the CFG a special vertex $trace_s$ carrying specific information about the AST node we are processing, and therefore about the source-code instruction that is being referenced by the AST. The $trace_s$ vertices are the only vertices that will be considered as observable actions when building the final LTS. All the other vertices will be treated as τ actions (see Section 2.9.3 on page 36).

- $label(s,m) = \{(s_{in}, m), (m, s_{out})\}$

  label creates a node m that can be accessed from a distant node in the CFG, thanks to the *jump* instruction.

- $jump(s,m) = \{(s_{in}, m)\}$

  jump breaks the normal sequential flow of a program, creating a link from the current node s, to a distant node m, defined with a *label* instruction. *label* and *jump* are used for the typical case of *Goto-Label* statements.

- $ctxlabel(s,m,s') = \{\}$

  ctxlabel does not add CFSL instructions to the DAST. The only effect of declaring contextual labels is that the necessary mappings $s \mapsto (m \mapsto s')$ are added to the DAST. Since this is already done by the *decorate* function, the *expand* function does not need to do anything else.

– $ctxjump(s, m, t) = \begin{cases} \{(s_{out}, t'_{in})\} \text{ if } t \mapsto (m \mapsto t') \in CD \\ ctxjump(s, m, parent(t)) \text{ otherwise} \end{cases}$

ctxjump recursively traverses the *ancestor* nodes of the current node s (starting from s itself), looking for some node t linked with m (by the *ctxlabel* instruction). The traversal is performed thanks to the *parent* function, therefore it is guaranteed that the first node t declaring m that is found, is the closest to s. Once t is found, the link is established from s to the node t' internal to t.

The *ctxlabel* and *ctxjump* instructions are commonly used when defining *Break* and *Continue* constructs working together with loops like *WhileDo* and *For*. The loop constructs use *ctxlabel* and *Break* and *Continue* use *ctxjump*. The third parameter, the state s' internal to s, used with *ctxlabel* is justified by the fact that, for instance, *Break* and *Continue* jump to different nodes inside the loop: *Break* jumps to the end (s_{out}), while *Continue* jumps to the beginning (s_{in}, or the state incrementing the counter, in the case of *For*).

4.4.3.3 CFG Reduction

Reducing the CFG is the process of removing, from the full CFG, those vertices that from a control-flow point of view, can be considered as irrelevant for the verification process. This reduction allows for a simplified solution to the problem of not being able to prove a full equivalence relationship between programs. By reducing the CFG we assess a less strict, or reduced, equivalence relationship. Even though this reduced program equivalence does not rule out all the possible problems related to program translation, it does verify the main issues regarding the execution order of sensitive instructions. It provides, thus, a non negligible increase in the user confidence degree of the translation.

CFG reduction serves two basic purposes that facilitate the equivalence verification process. First, it simplifies the graph and its transition system, minimizing the set of states we need to test. Second, it minimizes the risk of *local* incompatibility due to adaptive transformations. When we apply transformations to adapt the code, we modify the structure of the programs, generally by adding, deleting or moving some instructions. Checking for equivalence under these circumstances may fail in those places that where modified, even though globally the equivalence holds. The reduction is performed in two steps.

- The first, and most important step, is implicit to the definition of the CFSL instructions. This step is done by asking the user to mark in the grammar, with the `trace()` CFSL instruction, those significant constructs that must be checked for observation equivalence. For the case

of OLs those constructs are, besides the typical control-flow instructions, telemetries, telecommands, and certain specialised functions like directives from the mission control centre. All those constructs constitute the interaction with the satellite. The rest of the constructs, those that are not marked as traceable or observable, will be considered as hidden actions, and implicitly marked with the special symbol τ (see Section 2.9.3 on page 36).

- Second, the process of generating the CFG produces many arcs, which, even though necessary when building the CFG, are useless for our purposes of checking equivalence. More specifically, we do not need arcs linking the special vertices *in* and *out*.

The second step of the reduction, therefore, removes unnecessary *in* and *out* vertices using the function $reduce : CFG \rightarrow CFG$.

- $reduce(cfg) = remove(relink(cfg))$

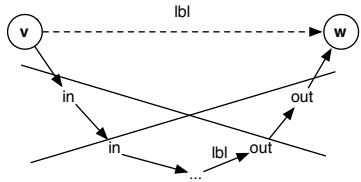
  reduce applies two auxiliary functions. First *relink*, that adds the arcs that will replace the connectivity provided by all the *in* and *out* vertices. Then *remove*, that eliminates from the CFG all the *in* and *out* vertices, as well as any related arc (by using the auxiliary $\ominus$ function).

- $relink(cfg) = cfg \oplus a_1 \oplus \ldots \oplus a_i \oplus \ldots \oplus a_n$
  where $\{a_1, ..., a_n\} = \{(v, l, w) \mid$
  v, w are not *in* or *out* vertices $\wedge$
  $\exists u_1, ..., u_k \in V_{cfg} : u_1, ..., u_k$ are *in* or *out* vertices $\wedge$
  $\exists$ sequence $(v, l_0, u_1), (u_1, l_1, u_2) \ldots, (u_{k-1}, l_{k-1}, u_k), (u_k, l_k, w) \in A_{cfg} \wedge$
  $$l = \begin{cases} true & \text{if } true \in \{l_0, ..., l_k\} \\ false & \text{if } false \in \{l_0, ..., l_k\} \\ \varepsilon & \text{otherwise} \end{cases}$$
  $\}$

  The goal of *relink* is to create a new arc going from vertex v to vertex w, for every sequence of more than one arc leading from v to w. All the vertices involved in the sequence, with the exception of v and w themselves, have to be *in* or *out* vertices.

  Regarding the label used by the new arc there are two possibilities. First, all the intermediate arcs have the ε label, in which case the label for the new arc is also ε. Second, there is one label in the sequence that is different to ε (true or false), which will be used.

  One label at most can be different to ε in such sequence of arcs for two reasons. First, even though a *true* or *false* label can be part of an arc between two *in* or *out* vertices, this is only because of how we build the CFG. Such label was indeed produced by some other vertex representing a boolean expression (and therefore different to *in* or *out*). Second, by the definition of the sequences we consider, we cannot have a vertex that is not *in* or *out* in the middle of such sequence.



- $remove(cfg) = cfg \ominus \{v \in V_{cfg} \mid v \text{ is } in \text{ or } out \text{ vertex}\}$

  $remove$ builds the set of in and out vertices in the CFG, and pass it to the $\ominus$ function.

- The function $\ominus : CFG \times \mathcal{P}(V) \rightarrow CFG$, removes a set $W \subset V_{cfg}$ of vertices, and all the arcs related with those vertices, from a CFG. $(V, L, A, s_0) \ominus W = (V \setminus W, L, A \setminus \{(v, l, w) | v \vee w \in W\}, s_0)$

  Finally, all the in and out vertices, as well as all the arcs considering any of these vertices, will be removed from the CFG.

Before dealing in the next section with the problem of program equivalence verification, let us go through all the steps of generating a CFG. Figure 4.23 presents the complete specification of a small language we have designed for the example. At the left of Figure 4.24, we have a program accepted by the language in Figure 4.23. This program will produce the AST at the right of Figure 4.24, which finally generates the CFG arcs at the bottom of the same Figure 4.24. In a last step, the CFG will be reduced to produce the CFG in Figure 4.25, where only 6 relevant vertices are left.

Start	**Block**	**IfThenElse**	**WhileDo**
Syntax (SDF)	**Syntax (SDF)**	**Syntax (SDF)**	**Syntax (SDF)**
Block -> Start {cons(`Start')}	(IfThenElse | WhileDo | Break | Continue | Com)* -> Block {cons(`Block')}	`if' `(' cond:Expression `)' `then' then:Block `else' else:Block `endif' -> If {cons(`IfThenElse')}	`while' `(' cond:Expression `)' `do' block:Block `enddo' -> While {cons(`WhileDo')}
Semantics (CFSL)	**Semantics (CFSL)**	**Semantics (CFSL)**	**Semantics (CFSL)**
1. (default());	1. (default());	1. (IN,cond); 2. (cond,TRUE,then); 3. (cond,FALSE,else); 4. (then,OUT); 5. (else,OUT);	1. (IN,cond); 2. (cond,TRUE,block); 3. (cond,FALSE,OUT); 4. (block,IN); 5. (ctxlabel('break',OUT)); 6. (ctxlabel('continue',cond));
Expression	**Command**	**Break**	**Continue**
Syntax (SDF)	**Syntax (SDF)**	**Syntax (SDF)**	**Syntax (SDF)**
`A` | `B` -> Exp {cons(`Expression')}	`CMD` `W` | `X` | `Y` | `Z` -> Com {cons(`Command')}	`break` -> Break {cons(`Break')}	`continue` -> Cont {cons(`Continue')}
Semantics (CFSL)	**Semantics (CFSL)**	**Semantics (CFSL)**	**Semantics (CFSL)**
1. (trace());	1. (trace());	1. (ctxjump('break'));	1. (ctxjump('continue'));

Figure 4.23: A small language for building a CFG.

4.5 Lightweight Program Equivalence Verification

Our ultimate goal of building the control-flow graph of programs is to test their equivalence, or more specifically the observation equivalence of a translated program with respect to the original one. In this section we introduce

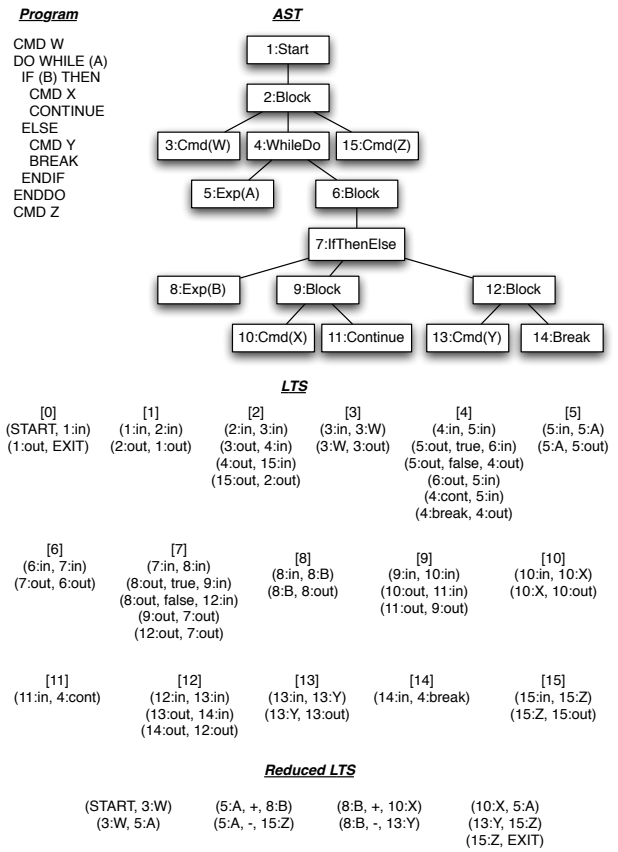


Figure 4.24: Generating the LTS.

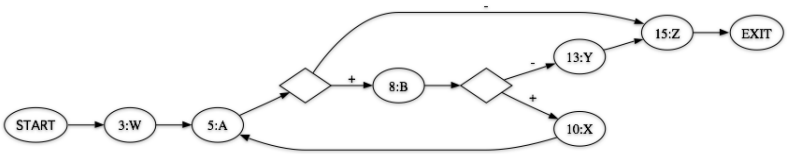


Figure 4.25: Final reduction to the graph.

a lightweight approach to verify program observation equivalence, based on the CFGs presented earlier in this chapter.

Our verification technique to assess the observation equivalence of a source program and its translated counterpart consists of four steps:

- First, produce the CFGs which models the programs' control-flow behaviour.
- Second, represent the CFGs as Labelled Transition Systems (LTS).
- Third, using a verification tool like the CADP Bisimulator [47], check if the two LTSs are equivalent modulo *weak bisimulation* (see Section 2.9.3 on page 36), also known as *observation equivalence*. If the two LTSs are *bisimilar*, we are done. If it is not the case, the verification tool provides a report pointing to the inconsistency. In this thesis we used two similar tools to confirm our results: the CADP Bisimulator, which is a solid and performant verification tool with many functionalities. We also implemented our own verification tool, which focuses specifically on doing weak bisimulation, and gives us additional control on the trace that is emitted as report. This is particularly useful when dealing with *non-strict tags*, as explained in the following paragraphs.
- Fourth, analyse the trace produced by the verification tool. We map this result to the CFGs and to the source code, marking the inconsistency in both the original and the translated programs, as shown in Figure 4.26.

At the top of Figure 4.26, we have two programs being tested for equivalence, that are not bisimilar. After testing for bisimulation, the CFG will show in light gray the last common nodes: up to this point the nodes for both programs are bisimilar. In dark gray we can see the first inconsistent node detected: a node is inconsistent if, from the last common node, it can be reached in one program but not in the other. At the left of Figure 4.26, an edge exists from B to D. At the right of the figure there is no such edge. Finally, in the code at the bottom of the figure, the tags *«Bisim»* delimit the source code fragment between the two shaded nodes that contain the inconsistency.

4.5.1 From Control-Flow Graphs (CFG) to Labelled Transition Systems (LTS)

Both representations, the CFG and the LTS, can be considered as abstract state machines. Both are used to model the behaviour of a program or a process, though at a different level of abstraction. We mention some of those differences relevant to our work.

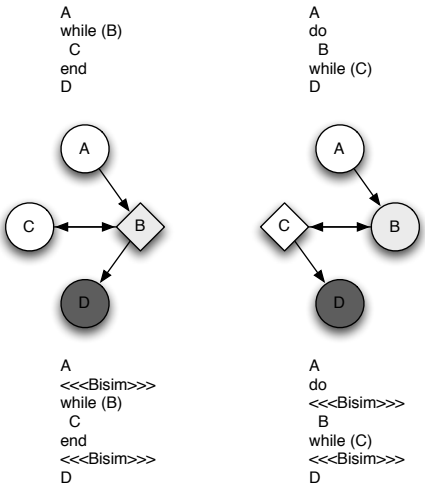


Figure 4.26: Showing to the user a bisimulation problem.

- According to the naming conventions, a CFG has nodes and edges, while an LTS has states and transitions. Nodes correspond to states, and edges to transitions.
- Tags or labels represent actions in both, though in a CFG they are normally in the nodes, while in an LTS they are in the transitions. CFGs can use additional tags in the edges, to express for instance when a path can be followed. LTSs consider the special silent action τ, unobservable from the outside.
- Graphically, CFGs tend to be more expressive, using different shapes for nodes having specific types of behaviour. For instance diamonds can be used for conditional nodes. LTSs do not make any distinction between states, except for the start state that, if necessary, can be drawn with a double line.
- In general CFGs have a more general use, being employed for complex code analysis and manipulation tasks. LTSs can be seen as specific to the field of process verification, and therefore many of the existing tools in the domain rely on the LTS representation.

Figure 4.27 shows the three representations of a program: source-code, CFG and LTS. Note that the `+` and `-` labels in the CFG represent the values *true* and *false* respectively: if `B` is evaluated to *true*, the path to `C` can be followed; if *false*, then we proceed with `D`. In the LTS this is represented with the labels `+B` and `-B` (if `B` would be considered as a silent action, we would keep anyway the + and - labels in those transitions).

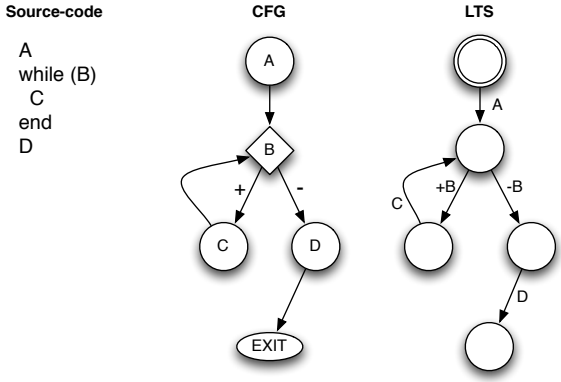


Figure 4.27: Source-code, CFG and LTS of a program.

The user has two alternatives when producing the LTSs for the bisimulation analysis. First, a *strict-tag* labeling can be used. In this case, the labels in the LTS include the unique id of the CFG node (and previously the AST node) that generated them. e.g. "13:Cmd1" means that the instruction *Cmd1* was present in the node *13*. The second alternative is the *non-strict-tagging* where only the name of the instruction is used. The CADP bisimulator tool uses only the text in the labels to establish the comparison, so these two mechanisms can produce different results, as depicted in Figure 4.28.

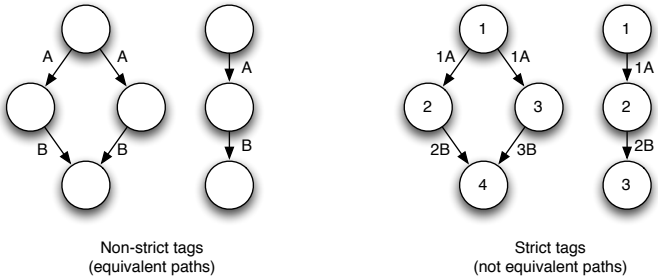


Figure 4.28: Strict vs. Non-strict Tags.

The example at the left of the figure uses *non-strict tags*, thus the two LTSs are equivalent.

$$A.B|A.B \sim A.B$$

The example at the right of the figure uses *strict tags*, and there is no equivalence between the two LTSs.

$$1A.2B|1A.3B \not\sim 1A.2B$$

Strict-tag labeling is better suited when we use automatic transformations, because we work based on the same AST. The unique identifiers are

preserved, and we can take profit of the additional precision in the bisimulation. When we want to compare two procedures that do not come from the same AST, strict tagging is no longer suitable: we cannot guarantee the same ids for equivalent AST nodes. This is the case when we manually modify a program, and later on we need to check if it is bisimilar with the previous version. Non-strict tagging must be used then, providing the standard bisimulation precision.

4.5.2 Checking Observation Equivalence

Verifying if two programs can be put in an observation equivalence relation, known as weak bisimulation, is a process separated from translation. From the usability point of view we can apply the verification technique to any two programs, independently of how they were generated, as long as we can get their LTSs. Some practical restrictions and requirements exist, though:

- There must be a one-to-one relationship, for every observable instruction (i.e. marked as traced), from source to target grammar. If the source language is capable to execute one of these instructions, so must the target language.
- Our verification technique is oblivious to side-effects in the execution of a program, unless the instructions involved, for instance logging instructions or assignments to variables, are marked as traced by the user. Otherwise they will be considered as silent τ actions during the weak bisimulation.
- We assume that transformations implemented to solve mismatches are reduced to the minimum necessary to ensure compatibility. If some transformations are implemented to reengineer or improve the code we are going to check, we cannot make any claim regarding their effect on observation equivalence.

Observation equivalence, also known as weak bisimulation, was formalised in 2.9.3 on page 36. In this section we provide a more intuitive view based on a simple example where we see how two programs P and Q in Figure 4.29, are verified for observation equivalence. For convenience the observable instructions marked as traced by the user, are simply denoted by capital letters, and the silent instructions by lower case letters.

First, we need to produce the LTSs for P and Q, and rename all the silent actions to τ. Then, we initiate the process from the start state of both LTS. The LTSs are shown in Figure 4.30.

We have numbered the states to make it easier to follow the bisimulation:
- We start by assuming that the start states are equivalent: $p1 \approx q1$,
- since $p1 \stackrel{A}{\Rightarrow} p3 \wedge q1 \stackrel{A}{\Rightarrow} q3$, we must then have that $p3 \approx q3$
(according to weak bisimulation, p2 and q2 do not need a bisimilar state, because they are part of a silent transition)

```
#P
A;

while(x) {
  C;
  D;
}

E;
```

```
#Q
A;
if(y) then {
  do {
    C;
    D;
  } while(y);
}
E;
```

Figure 4.29: Two observation equivalent programs.

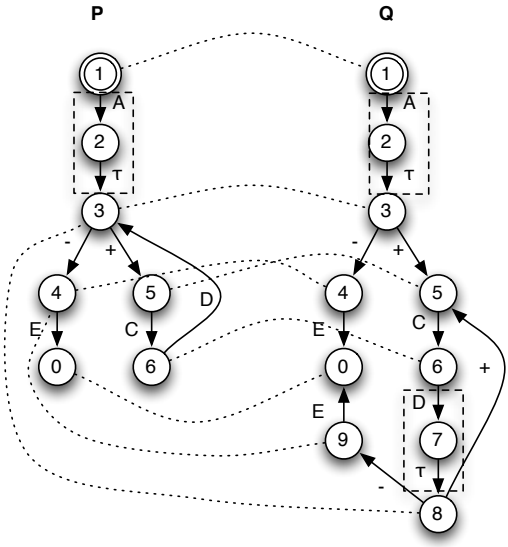


Figure 4.30: Two observation equivalent LTSs.

- next, since $p3 \stackrel{-}{\Rightarrow} p4 \wedge q3 \stackrel{-}{\Rightarrow} q4$, therefore $p4 \approx q4$,
and the process continues this way for the following states:
- $p3 \stackrel{+}{\Rightarrow} p5 \wedge q3 \stackrel{+}{\Rightarrow} q5$, and $p5 \approx q5$
- $p4 \stackrel{E}{\Rightarrow} p0 \wedge q4 \stackrel{E}{\Rightarrow} q0$, and $p0 \approx q0$
- $p5 \stackrel{C}{\Rightarrow} p6 \wedge q5 \stackrel{C}{\Rightarrow} q6$, and $p6 \approx q6$
- $p6 \stackrel{D}{\Rightarrow} p3 \wedge q6 \stackrel{D}{\Rightarrow} q8$, and $p3 \approx q8$
(again, q7 does not need a bisimilar state, and one state –p3– can be bisimilar with many states –q3 and q8)
- $q8 \stackrel{+}{\Rightarrow} q5 \wedge p3 \stackrel{+}{\Rightarrow} p5$, and $q5 \approx p5$
(note that any of the two LTS can lead the bisimulation, as long as it has transitions that have not been checked)
- $q8 \stackrel{-}{\Rightarrow} q9 \wedge p3 \stackrel{-}{\Rightarrow} p4$, and $q9 \approx p4$

- $q9 \stackrel{E}{\Rightarrow} q0 \wedge p4 \stackrel{E}{\Rightarrow} p0$, and $q0 \approx p0$
- Finally, no more transitions exist: $q0 \not\Rightarrow \ \wedge p0 \not\Rightarrow$, confirming that $P \approx Q$ ($s \not\Rightarrow$ means that there are no transitions from state s).

Thanks to the technique of weak bisimulation we can provide an additional degree of confidence on the observation equivalence of translated procedures. Moreover, when a lack of equivalence exists, the user is pointed to the exact place in the code where the problem appears.

The technique, however, cannot be considered as definitive for this kind of problems. Many limitations stem from the fact that we are making an imprecise abstraction when we define certain actions as silent. A good improvement would be to study how to combine this approach with data-flow and side-effects analysis techniques.

4.6 Conclusion

Automated support for translating procedures between different operations languages, in the domain of spacecraft mission planning, is an important issue. Firstly, there is an industrial demand for general-purpose tools that can manipulate and translate between procedures in any of the operations languages that currently exist. Secondly, the current tendency to strive towards a standard operations language strengthens the need for translation tools, not only to translate procedures in old operations languages to the new standard language, but also because there is a need for translating between the two languages competing to become a standard.[5]

Although the language translation problem is, in general, a very hard problem to tackle, for the case of operations languages the problem is easier because they all share a common semantical basis. This makes it feasible to semi-automatically generate transformation tools that can automatically translate procedures written in one operations language to another operations language. More specifically, we have shown how annotated grammar definitions can support this automated generation of translators between languages. The annotations are used to define a mapping between corresponding grammar productions. From these mappings, rewrite rules that transform the corresponding productions can be derived automatically. The translator thus produced only needs to be completed to handle those productions for which no mappings were given.

We have shown how to augment the annotations with more semantic information on the flow of control in the grammar productions. We illustrated how these additional annotations could be used to produce a verification tool for the correctness of the translation process.

[5] At the time of writing, the *PLUTO* language [42] was being put forward as a standard by the ECSS (European Cooperation for Space Standardization), whereas the language underlying the *MOIS* system [95] has been proposed as a standard to the OMG (Object Management Group).

The approach for program equivalence verification presented in this thesis is an interesting starting point for more advanced experiments on the use of bisimulation techniques for verifying program translation. Moreover, we believe that with the right set of advanced annotations, an early assessment prior to the translation can be provided, based only on the annotated grammars and annotated transformations.

We conjecture that the advanced annotations can also be used to enhance the automated generation of program translators beyond simple one-to-one mappings between grammar productions. These last two extensions are part of our ongoing research work.

5 A Product-line Approach

Previous chapters, particularly Chapter 4, introduced and described our approach for building program translators between two different languages. In this chapter we extend the approach to families of languages.

Let us consider the example family of Operations Languages, OLs, employed by space industry to design procedures used to test and control spacecrafts. In general, programs built with these OLs describe high-level, goal-oriented activities to be carried out by a spacecraft. These high-level activities or procedures are built up of more elementary activities such as *directives*, which are functions or procedures executed directly by the control centre, *telecommands*, which are instructions uploaded to the spacecraft to execute an action, and *telemetries* which are blocks of data received from the spacecraft as a measurement of its current state. Different circumstances can require users of these procedures to translate them to a new language, for instance due to technology change, the advent of new standards, or company merging. A concrete example for this case can be found in [87] where a consultant company for the space industry is confronted with the need of a program translation solution for its procedure design tool.

The case of OLs is not unique nor special. Program translation is not an uncommon need since programming languages typically evolve with industry. New functionality gets added, some gets deprecated, and languages keep growing in new versions. Languages also have a life span. When they stop being maintained, eventually they become legacy languages. On the other hand new ideas and technologies emerge, and with them new programming languages. This evolving scenario produces many examples like that of OLs. We have already mentioned some of these examples. For instance in [28], Cleve presents a case inside the data-base community, where a translation approach would have been a feasible alternative in a software migration project for database query and manipulation languages.

For the general problem of program translation there is no universal solution that can always be applied. Studies like those from Verhoef et al. [65] and Terekhov et al. [114] show that automated language conversion remains in practice hard to achieve, partly because of the many semantic differences between languages. Nevertheless, using the appropriate technology, like for instance by applying reengineering techniques to the code before translating it, it is possible to reduce the complexity of the problem, and to build specific automated translators. This approach can be useful specially for some specific application domains where programming languages are semantically related and can be grouped in families. By limiting the approach to languages

belonging to the same family, advantage can be taken of structural and semantic similarities between those languages to semi-automatically build a family of translators between them.

Building program translators between a group of languages sharing many similar properties, but at the same time manifesting certain differences, requires an answer to the question of how to take maximum profit of commonalities among languages, and therefore among translators, and how to minimise the overload due to solving language differences.

Our solution follows a product-line engineering approach. This approach strives for reuse, and defines how to take profit of similarities between products by explicitly defining how their components are related. Commonalities and variabilities between products are therefore explicitly taken into account during the design phase of a product-line, and reuse of components is maximised when building the products. A general description of product-line approaches can be found in Section 2.10. In the current section we will instantiate the product-line concepts for the specific case of building families of program translators.

A Translation Schema.

Our product-line solution extends our language-to-language translation approach, explained in the previous chapter, to families of languages and translators.

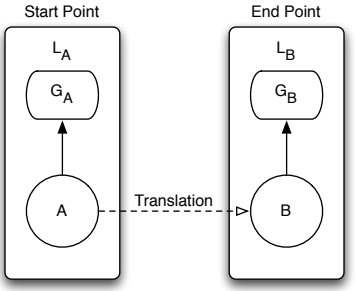


Figure 5.1: A translation schema.

Figure 5.1 shows the basic schema of a translation with two end points. The *Start Point*, to the left of the figure, represents the input for the translation. The original program A or *Source Program* is accepted by the grammar G_A, or *Source Grammar*, which represents the language L_A or *Source Language*. The *End Point* represents the output of the translation. More specifically, the program B or the *Translated Program* is accepted by the grammar G_B or *Target Grammar*, which represents the language L_B or *Target Language*. Theoretically, a new translator needs to be built for each pair of source and target language.

A Product-line Model.

Our instantiation of the product-line model for families of languages and translators is defined in Figure 5.2, showing the main parts of this product-line: scope, core assets and production plan.

- The scope defines the family of languages we are interested in and the translators we are going to build.
- The core assets, derived from the scope definition, are the different elements we are going to put together: the tools we need to build the translators; the languages, represented by their grammars and language concepts, and the transformations that will adapt the programs during the translation.
- The production plan describes the different processes involved in the production of the translators: recovery of a working grammar when it is not available, convergence of all languages into a central family structure, and finally generating and testing the produced translators.

This product-line model is a rather technical way of describing our product-line. To harmonically structure all these elements such that they are defined and added to the product-line in the correct order, we have divided the building process in four production phases, each of which are detailed below.

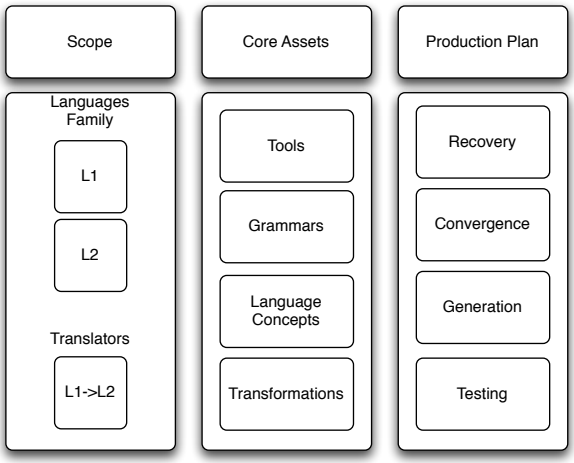


Figure 5.2: The product-line model.

Production Phases.

Our ultimate goal is to translate programs. The complete production process of our product-line, to go from programming languages to translators be-

tween these languages and translated programs, consists of four main phases, sketched in Figure 5.3.

The first phase, scoping the product-line, defines all the elements that will be part of it: languages, translators, etc.

In the second phase, structuring the product-line, we design and build the product-line structure. This is a manually intensive phase where the product-line designer needs to get all the core assets, process them, and put them together into the product-line structure.

The third phase, translator generation, is an intermediary automatic step, where all the required translators are produced and ready to be tested. We will present this phase together with the last phase.

Finally the testing phase consists of evaluating the translators with a set of programs, checking if the original and the translated programs are functionally equivalent. If the results of the testing phase show some unacceptable differences, the designer will have to modify the generated program transformations and retest them. If the designer cannot provide a solution for every incompatibility between languages and language concepts, a partial translation may still be produced. The rest of the chapter elaborates on each phase.

5.1 Scoping

Defining the scope of our product-line ultimately consists of defining the set of translators that will be generated as final product. The scope thus is defined by construction, and is dependent on the group of languages that will be targeted by the translators. The scope definition tends to be driven by business needs, rather than by technical considerations. Unfortunately, from the business point of view, the necessity of having a certain translator does not imply its feasibility. Even more unfortunate, the unfeasibility of translating programs between two given languages often does not show up early enough in the process of building a translator. Acquiring some confidence on the viability of the translators as soon as possible, will save valuable time and effort. Gathering enough information on the languages we are including in our family is essential to increase this confidence.

Regarding the extent of the information we need, it is desirable to start the scope definition with as many languages as possible. Technically, there is no restriction to initiate a language family with as few as two languages. Including other languages iteratively as it becomes necessary to build new translators is possible. The inconvenience of such an iterative process of growing a family, however, manifests itself at the reusability level, where translations implemented on the basis of just a few languages risk to be more difficult to reuse.

In depth, the scope needs to consider not only the languages, but also the language concepts beneath. We consider language concepts as first-class

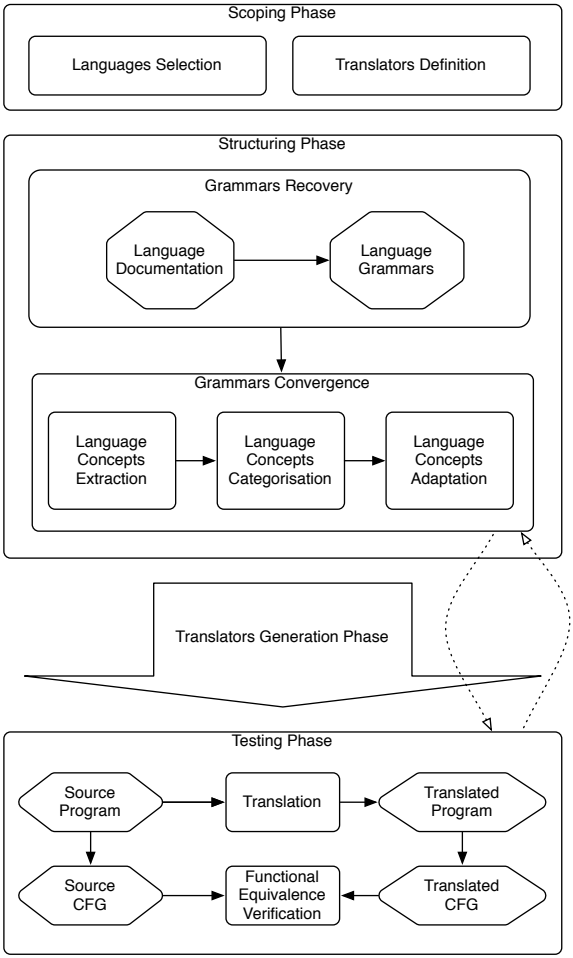


Figure 5.3: Product-line production phases.

elements in the family, and not limited or restricted to a single language. It is during the scope definition that we set up the basis for the final structure of the languages family.

How we build the scope of a language family can be summarised by the following three questions:

1. What are the members of the language family? This apparently trivial question defines indirectly the application domain. Even if there is no restriction on the number of languages, or if the product-line can be extended afterwards with additional languages, it is important to establish our starting point, which provides the basis for the rest of the process. As we have already mentioned, even if it is possible to start the product-line with as few as two languages, the more languages we have for the initial scope definition, the better the general knowledge on the language domain we will acquire, and the more effective our efforts to adapt language mismatches.

2. How similar are the languages in the family? Answering this question requires an in-depth knowledge of every language in the language family. We need to know their internal structure to compare all the languages based on the features and concepts they offer.

3. How complete can we expect the resulting *automatic* translators to be? Based on the information obtained from the previous questions, we have a preliminary idea of incompatibility issues, and we already know where to focus our efforts. Thanks to this question, we focus precisely on what has been found as directly incompatible. We try to define precisely how the mismatched language structures in one language, can be reduced to different language structures in other languages, becoming indirectly compatible.

These questions point directly to avoiding at an early stage two major risks in practice regarding product scoping. If the scope is too large because we try to include too different languages in the family, then accommodation to variability will be hampered by the impossibility to adapt language concepts. If on the other hand, the scope is too small, in other words if we have too few languages, the adaptions for every translator may not be generic enough to provide reusability [34, 100, 89].

The answers to these questions are provided by two interrelated elements that are part of our technique for structuring the language family. First, the Language Concepts Matrix (LCM) that relates languages to concepts, and second a set of language metrics that, when applied to the LCM, makes evident the nature of those relationships. Detailed explanations for the LCM and the set of metrics respectively are provided in sections 5.2.3.1 and 5.2.5.

Figure 5.4 shows an example of an LCM where the languages and their constructs are related, and from which some equivalences and relations between languages and constructs can be inferred directly. One the most im-

	Lang1	Lang2	Lang3	Lang4
Concept1	X	X	X	X
Concept2	X	X	X	X
Concept3		X		X
Concept4				X

Figure 5.4: The LCM table.

portant relations we try to establish is direct compatibility between language concepts and languages. For a language concept, being directly compatible with a language means that the language provides a native construct whose abstract syntax structurally corresponds, and whose functional semantics is equivalent to the language concept.

For instance, concepts 1 and 2 are compatible for all languages, while the other two constructs are only partially compatible. Language 4 can be considered as a universal receiver in a translation because no adaptation will be needed when translating from the other three languages to this language, since language 4 has all the concepts provided by languages 1 to 3. On the other hand, translating from language 4 to any of the other languages will require implementing some adaptations. For example, construct 3 needs to be transformed when translating to languages 1 and 3, and construct 4 always needs to be transformed into native constructs of any other language we are translating to. When such an adaptation is possible, we can say that the language concept is indirectly compatible with the language.

For a toy LCM example like this, the relations between languages and constructs are easily perceived. For more realistic cases it will be a lot harder to see because of the number of languages and in particular because of the number of constructs in those languages. The purpose of our set of language metrics is to help us overcome this difficulty by providing a global view of the family and by pointing to specific places that can be either opportunities to take profit of (e.g. concepts shared by many language), or problems that need to be solved or avoided (e.g. concepts appearing in a small number of languages).

5.1.1 Core Assets

In this section we briefly summarise the considered assets in our product-line. All of these elements already have been introduced and explained in detail in the previous chapters.

Four kinds of core assets are distinguished in our product-line model: tools, grammars, language concepts and program transformations.

Tools are used to process the grammars and their language concepts, to generate the program transformations, and finally to generate and test the translators.

Grammars are the main input for defining our product-line. They de-

fine the languages, provide the language constructs, recognise the input programs, and validate the translated programs.

Language concepts are extracted during scope definition, from the grammars describing the languages. Language concepts are the main elements in the LCM table, and they are extracted from the language constructs, which are in turn extracted from the grammars.

Program transformations are the glue among mismatched constructs. These transformations are small, dedicated programs used to solve specific compatibility problems among languages. In general, languages considered inside a family share a large amount of language concepts. Moreover, our approach is restricted to those families with high levels of (syntactic) compatibility among languages. Nevertheless there always exist language incompatibilities among languages that need to be solved to make the approach useful: direct full compatibility between two given languages is very rare to find. Program transformations allow to adapt incompatible constructs in the source language, to some combination of constructs in the target language, providing functional equivalence.

5.2 Structuring

Building a translator with our approach, as was already pointed out in Figure 5.1, is a syntax-dependent process. The grammars of source languages allow to parse and manipulate the source code of the original programs. The grammars of target languages allow to unparse the transformed programs and produce the translated source code.

Our approach relies on the existence of working SDF grammars for every language to be included in the family. We do not address, however, how to build a grammar for a language. For completeness though, we consider in our approach the possibility of not having direct access to these grammars. In the following paragraphs, we provide some information on possible ways of circumvent such cases, notably with the use of grammar recovery techniques [64].

5.2.1 Recovery: Obtaining the Working Grammars

The grammars we require, to build the product-line, are not always directly or easily available. This availability depends on the accessibility of the languages we are working with, which depends a lot on the specific application domain. For our specific case of OLs, Pluto and UCL provided a complete BNF grammar in LL style –scattered among other information, for Ucl. Mois provided an XSD grammar. Elisa and Tope had an informal BNF grammar linked to examples, and for Stol there was only examples of how to use the more relevant instructions.

Availability of the grammars comes in different degrees. In the best case, which is the assumed case in this dissertation, is to have a directly available

working SDF grammar. Not ideal, but still a good alternative, is having working grammars but in a different format, like ANTLR or YACC for instance. The worst situation is not to have any grammar at all.

Assuming the worst case, when grammars are not available, we must at least count on some sort of documentation, like manuals or examples, that can be used to extract or infer these grammars from. The specific process of extracting working grammars from various sources of information has been called Grammar Recovery, and has been addressed by Lämmel et al. in [64]. Grammar Recovery, in conjunction with the Grammar Deployment Kit [60] can significantly facilitate this process. Unfortunately this step remains highly manual. A-priori we have no control over the kind of information we are going to receive, and a considerable amount of effort could be required in reconstructing the required SDF grammars.

Using the Grammar Deployment Kit must be considered only as a recommendation, and it is not mandatory for this step. This set of tools, for instance, can be really useful when we already have a grammar but in a different format, like Yacc or JavaCC. As the creators of the technique say in [64] "...there are no real restrictions on what technology to use, so go ahead and use your favourite transformation tool, ... so you can quickly produce the parsers and other grammar-based tools that you need."

From the moment we have obtained the required SDF working grammars, we can initiate the core of this phase, which is explained in the following two sections –5.2.2 and 5.2.3–, where we adapt some of the ideas of Grammar Convergence [66] into a product-line approach for building program translators.

First, we prepare the grammars in a process analogous to normalisation, such that the language concepts underlying the constructs in the grammar can be easily extracted, as independently as possible of other languages in the family. Then we use the language concepts as properties of the family, and categorise every language according to the properties they present. We relate this way languages and language concepts into a single language family structure. From this structure, we can efficiently build the translators.

5.2.2 Language Concepts Differentiation

Our technique for relating languages and language concepts relies on some of the syntactic properties of the constructs provided by the languages, like their symbols and keywords. These language-specific syntactic properties, we believe, relate with the more generic semantic properties of the language concepts underlying the constructs.

Together, syntactic and semantic properties, constitute the structural properties of languages. Classifying languages according to their structural properties is strongly related to *Language Typology*. Language typology, as defined by Caffarel et al. in [75], is, in a broad sense, the general study of similarities and differences across languages. Among the many existing typo-

logical approaches, Whaley encourages the use of those approaches involving the classification of languages –and elements of languages– based on their shared properties [130].

This category of approaches consists of first directing the attention towards a particular construct that arises in a language. Then, using cross-linguistic data, all the possible types of this construct in other languages are determined, thus inferring a category based on this shared property. Even if these typological approaches mentioned by Whaley, were originally conceived with natural languages in mind, they fit remarkably well to our needs because of their structured nature which matches our grammar-oriented approach.

From the working concrete grammars successfully used for parsing and unparsing source code, we can extract an abstract representation of every language where only the relevant structural elements have been retained.

From this condensed abstract representation we can infer two things: common language concepts shared among multiple languages that will be put immediately together, and variations specific to some languages, that will be adapted through program transformations.

Getting the abstract, conceptual representation of the constructs in a grammar requires some pre-processing. A concrete grammar that successfully parses programs has good probabilities of not being conceptually clean, because of practical constraints imposed by the interpreter or compiler of that language. An analogy with database designing can be made: sometimes a clean, normal-form design of a database, needs to be de-normalised to meet the requirements of a specific database engine.

Extracting Language Constructs from a Grammar The first step to classify the constructs coming from some language, inside our languages family, requires the extraction of the abstract concepts from a concrete language grammar. Language constructs are physically represented by means of productions inside a language grammar. The way these productions are designed vary from language to language. For the same construct we can have a different syntax in two different languages. Extracting, or differentiating the productions representing an instance of some language construct is a non-trivial problem.

In this section we present a lightweight methodology for differentiating and extracting the different language constructs from a grammar. This methodology is composed of guidelines, or rules of thumb (RoT). This set of guidelines was extracted from the experience obtained with our case study (see Chapter 6). They worked very well during our experiments, and we have good reasons to believe that they will be applicable and useful for other families too.

Although we warmly suggest the user to follow these guidelines, there are exceptions to its use. When possible we present examples of those exceptions. For application domains with noticeable differences with the domain from our case study, discretion is advised when using these rules.

RoT#1: Prefer grammar modifications over program transformations. The family structure and the resulting mappings will be cleaner.

As mentioned before, in this work we are not interested in how to build the initial grammars of the languages we receive as input. We assume we already have valid SDF grammars for all languages of the family. These grammars, nevertheless, should be susceptible of being modified along the process of classifying the languages.

When some part of a grammar requires some adaptation to allow for a better mapping with the rest of the languages of the family, we have two alternatives. First, to modify the original grammar. Second, to build a set of program transformations that modify the resulting ASTs of the concrete programs we receive. In our approach we give preference to the first alternative. It is not always possible to come up with an equivalent grammar for some set of productions, but it is the cleanest solution: transformations tend to clutter the structure of the language family.

RoT#2: Extract the constructs top-down. A systematic approach for analysing the grammar gives a clean start to the process.

There is a big number of productions in almost any programming language grammar, regardless of the grammar formalism used. Moreover, if we consider specifically the case of our SDF grammars, there is no constraint on the order in which the productions are defined. Dependencies among productions do not impose any restriction on whether some production needs to be defined before or after another. All this makes it difficult to know where we should start the analysis for extracting the language constructs.

Two important things are known, though. First, the relevant productions are to be found inside the `context-free syntax` section of the SDF grammar. The rest of the sections can be avoided. Second, and more important for this rule of thumb, the `start-symbols` section defines what are the symbol, or symbols, that according to the grammar designer are the accepted start symbols for the parse trees. Moreover, for a majority of the languages, there is one single accepted start symbol, referencing the full program or procedure.

Even though one may be tempted to start picking up from the grammar those productions that can be recognised at first sight, that approach provokes too many extra revisions of the grammar, and is therefore not advised.

Extracting constructs starting from the start symbol, and then going top-down as organised as possible, proved to be the best alternative during our case study.

RoT#3: Try to avoid lexical syntax symbols. Even though lexical differences have to be solved later on in the translation, the symbols that define lexical syntax should not be considered as language constructs in this step.

Let us consider the following example, where we define identifiers for some programming language. An identifier is an abstraction that represents a

variable that points to the value inside some memory location.

```
[a-zA-Z_] -> AlphaChar
[0-9] -> NumChar
AlphaChar (AlphaChar | NumChar)* -> Identifier
```

Although three symbols are present in the example, the only one that will be promoted to language construct is the symbol *Identifier*. The symbols defining the lexical syntax, *AlphaChar* and *NumChar*, are only auxiliary symbols used to recognise characters to build a valid identifier. If we take a closer look at the example, we can see that we do not even need the auxiliary symbols, because they can be directly injected in the definition of *Identifier*.

To reinforce the idea that the user should analyse every case from a semantic point of view, looking for structures with a complete meaning, let us assume that *Identifier* was defined without any auxiliary symbol.

```
[a-zA-Z_] ([a-zA-Z_] | [0-9])* -> Identifier
```

Now, *Identifier* can be considered as a lexical symbol as well. What probably seems ambiguous, is that we are promoting it to a construct anyway. Our example becomes now an exception to the guideline.

Let us clarify these things. The main consideration for promoting *Identifier* as a language construct, and not *AlphaChar* or *NumChar*, is that in this context *Identifier* represents an entity with a well defined semantics. It is a required abstraction whose definition cannot be simply injected into other symbols, without loosing expressiveness. In general lexical symbols are there to help building more complex definitions. Nevertheless, exceptions are always present, depending on how the grammar was built.

RoT#4: One construct can span several productions. A single language construct can be defined in more than one production.

First, do notice that in our previous rule of thumb we already had a construct that needed three productions to be completely defined.

This case however is different. Our example is related with how grammars for LL parsers are designed to avoid left recursion. The operation in the example is the arithmetic addition. For a GLR parser this operation can simply be defined as `Exp "+" Exp -> Exp {left}`, recognising expressions like `1 + 2 + 3`, and linking them from the left. The case for an LL grammar is more complicated. An LL grammar will not show a straightforward definition, and the user will require extra attention to understand the productions involved. Let us see the example.

```
Head Tail -> Addition
NUMBER -> Head
"+" NUMBER Tail? -> Tail
```

The language abstraction that interests us here is the arithmetic addition, let us call it *Add*. The first reaction is to consider only the *Addition* symbol

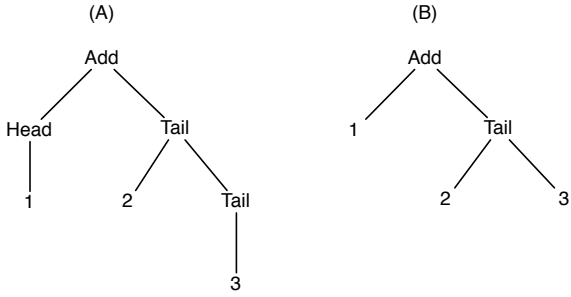


Figure 5.5: Parse trees for the 1 + 2 + 3 expression.

as defining the construct, and to leave away *Head* and *Tail*. For the case of *Head*, it is indeed an auxiliary symbol. We can get rid of it, and inject *NUMBER* directly in the definition of *Addition*. The case of *Tail* is more complicated. Even though it looks like an auxiliary symbol, it is not. It is complementary to *Addition*, and it has the same meaning. Therefore both, *Addition* and *Tail*, represent the same *Add* construct. In Figure 5.5 we can see the tree generated by these productions, for the expression `1 + 2 + 3`.

In (A) we have the more complete parse tree. If we simplify the tree, getting rid of the unary branches roots, we have in (B) a more succinct representation. As you can see, if *Tail* is not annotated as the *Add* construct, the sub expression `2 + 3` will not be recognised as an addition.

An even better and cleaner solution for this specific example would be to reformat the grammar into a GLR style: `Exp "+" Exp -> Exp {left}`. Our next rule of thumb illustrates why this is a better alternative.

RoT#5: Prefer a high-level grammar style. Avoid the implicit encoding of associativity and precedence in the grammar.

This case, like the previous one, relates with problems we can find inside LL grammars style. It mainly relates with how LL grammars tend to manage operator precedence and associativity. Let us analyse a small example of how a parser generator like ANTLR handles precedence between a '+' addition operator (AddExp), a '*' multiplication operator (MultExp), and a '^' exponentiation operator (ExpExp). The precedence order, from higher to lower is: '^', '*', '+'. Addition and multiplication associates from the left, and exponentiation associates from the right. We present the example using SDF syntax to ease visualisation and comparison.

```
MultExp ("+" MultExp)* -> AddExp
ExpExp ("*" ExpExp)* -> MultExp
NUMBER ("^" ExpExp)? -> ExpExp
```

This grammar recognises an expression like: `1 + 2 * 3 ^ 4` as if it would have been parenthesised like: `(1 + (2 * (3 ^ 4)))`

This is a nice example of explicit encoding of precedence and associativity rules. First, the precedence is encoded thanks to the nesting of the productions. The deeper in the hierarchy, the higher the precedence. Second, the direction of the associativity is encoded thanks to iterations for the left associativity (AddExp and MultExp), and tail recursion for the right associativity (ExpExp).

From our point of view focused on the translation based on mappings, the main problem of using this approach, is in the extreme interdependence of the symbols. Let us suppose we have the very simple expression `1 + 2`. What we can intuitively expect as AST from this expression is as depicted in Figure 5.6(A). What is really produced by the parser is definitely more tangled, as shown in Figure 5.6(B).

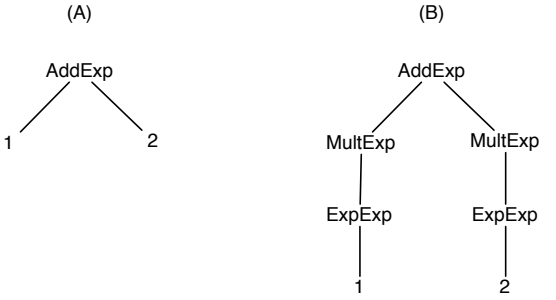


Figure 5.6: An AST with implicit encoding.

The AST we obtain with this grammar needs post-processing with some transformation to reduce the unnecessary symbols MultExp and ExpExp. The user should be aware that these symbols are unnecessary only in this context. For other cases, they are completely valid symbols. This makes the required transformations to adapt this tree, rather cumbersome because of the additional checking.

If we modify this grammar, and instead use a GLR style with precedence and associativity rules, as shown immediately, we will get the expected result.

```
context-free syntax
Exp "+" Exp -> Exp
Exp "*" Exp -> Exp
Exp "^" Exp -> Exp

context-free priorities
  Exp "^" Exp -> Exp {right}
> Exp "*" Exp -> Exp {left}
> Exp "+" Exp -> Exp {left}
```

Cleaner ASTs, and therefore cleaner mappings can be obtained thanks to the power of full context-free grammars, like SDF.

Do notice that the `context-free priorities` section does not need to be considered when looking for constructs. Only the productions inside the `context-free syntax` section are needed for that task.

RoT#6: Avoid productions with too much structure. Splitting functionality into auxiliary productions can make it easier to aligning grammars.

For this example, let us analyse the *IfElseIf* construct. Intuitively, this construct is composed of a list of blocks of code. Each one of these blocks is governed by a conditional expression. The first block whose conditional expression evaluates to `true` is the only one that will be executed. If no conditional expression evaluates to `true`, the construct can have a default block that, if present, will execute.

We can define the full construct with a single production, as follows:

```
"if" Expression "then" Block
("elseif" Expression "then" Block)*
("else" Block)? "endif"  -> IfElseIf
```

Even though this is completely valid, there is too much information for only one production. Good formatting, as in the example, helps to a better understanding of the production but there are bigger issues than just reading the grammar. Let us focus on the second line of the example. This part of the production represents a list of unknown length, whose elements are blocks of code governed by a conditional expression. If we leave the production as shown in the example, the instances of the list will be anonymous. Considering we base our technique in mappings against equivalent structures, and that we identify these structures by name, anonymous structures have to be avoided. Anonymous structures create an ambiguity problem. Overcoming this problem requires to implement conditional checking to know we are mapping the right structures.

If we consider a refactoring of the previously shown *IfElseIf* production, a possible alternative is the following.

```
Then ElseIf* Else? "endif" -> IfElseIf
"if" Expression "then" Block -> Then
"elseif" Expression "then" Block -> ElseIf
"else" Block -> Else
```

The definition now is more modular and easier to understand. We have differentiated three main components, and we have an auxiliary symbol for each one: *Then*, *ElseIf* and *Else*. More important though, going back to the problem presented in the previous guideline, this allows every instance of the *ElseIf** list to be named. Mapping these instances can be made directly, thanks to the auxiliary productions.

The symbol *Else* provides the same benefits as the *ElseIf* symbol. Regarding the last auxiliary symbol, *Then*, it is more a choice of the designer. It is not a required symbol, because it can be directly injected into its parent *IfElseIf* symbol, and semantics will not be affected.

RoT#7: A single production can represent more than one construct. One construct must have one semantics.

This is probably one of the more complicated situations. It is hard to detect because probably there is nothing in the grammar that warns about the problem. It is hard to solve because it will certainly imply a refactoring of the parts of the grammar involved.

Let us consider the *Wait* statement, coming from the Pluto operation language, that is defined in a single production in the reference grammar. In the production we have removed the portions of the syntax not required for understanding the example.

```
"wait until" Expression -> Wait
```

This production has three different semantics, depending on the value of *Expression.*, that rely on a hidden test, made by the compiler, that is not noticeable from the grammar. This test has a different nature, depending on the kind of Expression received by the Wait construct. Let us see the three different semantics of the test:

If Expression produces a *Boolean Value*, then the program waits until its value is `true`.

```
wait until Boolean Value == true
```

If Expression produces an *Absolute Time Value*, i.e. a specific moment in the future, then the program waits until the current time is bigger than the absolute time received in the expression.

```
wait until Absolute Time Value < Now
```

If Expression produces a *Relative Time Value*, for instance a number of seconds, the program waits until this amount of time elapses.

```
var := Now.getSeconds
wait until Relative Time Value < (Now.getSeconds - var)
```

Other languages are not so general as Pluto, and do not consider the three alternatives. This means that we cannot consider this case as a single construct, but as three different constructs with similar, but not the same semantics

The recommendation here is: reformat the grammar and create three different *Wait* productions, one for each semantics.

```
"wait until" AbsTimeExpression -> Wait {WaitAbsolute}
"wait until" RelTimeExpression -> Wait {WaitRelative}
"wait until" BooleanExpression -> Wait {WaitBoolean}
```

Remember that we heavily trust on the generated AST for the translation process. The AST therefore must contain as much semantic information, and the information should be as clear as possible, to give preference to simple mappings over complex transformations. Monolithic constructs, with several possible meanings, force to implement additional operations to extract the specific semantics concerned.

5.2.3 Language Concepts Categorisation

One of the pillars of our approach is the assumption that inside a well conceived language family, similarities between languages fairly outnumber their differences. We expect that for any two languages we compare, a big majority of the constructs in one language can be directly mapped to an equivalent construct in the other language. Moreover, for those constructs without a direct mapping in some language, we assume there exist an adapted mapping thanks to some program transformation.

To avoid mismatches between languages due to grammar and syntactic styles, the previous step presented a set of guidelines to uniformly extract constructs from a grammar, without requiring a step by step comparison with other grammars. By better regulating the shape of the constructs, we provide more uniformity between language structures, on a language-independent basis.

However, the relationships between language constructs, even if we adhere to the rules of thumb given in 5.2.2, are not easily seen. We need to organise languages and constructs in categories such that the different commonalities and variabilities between them are pointed out and emphasized.

By including categories in our system, commonalities –equivalent constructs– are mapped together directly when included into the same category. Variabilities –mismatches between constructs– can be adapted with a reduced number of program transformations: transformations can be built via shared constructs.

Our technique is related with Mosses' proposal of using reusable components for describing programming languages [79]. Mosses points out that different languages often have many constructs in common, even if they belong to different language families. These common constructs can be defined in a common repository. Every language reuses existing definitions and contributes with new definitions that can be later on reused by other languages. Next, we show how we elaborate and apply these ideas to the family of operations languages.

5.2.3.1 The Language-Constructs-Matrix, LCM

The Language-Constructs-Matrix, LCM, is a two dimensional matrix where the global relationships between languages and constructs are highlighted. A small introduction to the LCM was given in Section 5.1. In this section we provide an extended explanation based on an example extracted from our main case study (presented in Chapter 6).

The structure of the LCM is not complex. The columns represent languages, and the rows represent constructs. The cells or intersections between rows and columns, represent relationships between a language and a construct. Three kinds of relationships can hold:

- **Direct**. The language supports the construct natively. Specific productions in the grammar provide the syntax for the construct. The

construct's semantics and abstract structure defined by the language, are guaranteed to be completely equivalent across all the languages in the table defining the same construct. We represent this relationship in the table with a value of *true* (*X* in our examples).

- **Adapted**. The language does not support the construct natively. No productions exist in the grammar defining the structure of the construct. Nevertheless, a program transformation has been provided adapting the construct, when it is received from another language in a translation. This relationship is represented with the unique *Id* of the transformation performing the adaption.

- **None**. No direct support exists for the construct and, so far, no program transformation has been provided that adapts the construct functionality into something the language understands. A value of *false* (or *empty*) represents this situation.

Figure 5.7 shows the LCM for a subset of languages and constructs. This small, though non-trivial, example will be used along this chapter to describe the global data structures used to build a language family. Only *Direct* relations are shown in the matrix from Figure 5.7. Later on, we will introduce *Adapted* relations to the table, and explain how to reuse them.

	Stol	Tope	Ucl	Mois	Pluto
WaitUntilTimeAbs, WA	X				X
WaitForTimeRel, WR	X	X			X
WaitUntilBool, WB				X	X
RepeatUntil, RU	X	X	X	X	X
Assign	X	X	X	X	X
Add	X	X	X	X	X
Sus	X	X	X	X	X
Id	X	X	X	X	X
Gt	X	X	X	X	X
FuncTimeAbs2Rel, Fa2r	X	X	X	X	X
FuncNow, Fnow	X	X	X	X	X

Figure 5.7: An LCM table.

The example takes into account five languages: Stol, Tope, Mois, Ucl and Pluto, and the subset of constructs related with the *Wait* functionality. The *Wait* functionality groups some related constructs that stop the execution of the procedure temporarily, until some condition becomes true. We recognise for this example three variants:

- **WaitUntilTimeAbs**, or *WA* for short. The WA construct receives as parameter an expression *Exp* representing an absolute time value. The result of evaluating WA is that it stops the execution of the procedure, until the current time is greater than the value represented in

the expression Exp. This construct is directly supported by Stol and Pluto.

- **WaitForTimeRel, *WR.*** The WR construct receives as parameter an expression *Exp* representing a relative time value in seconds. WR stops the execution of the procedure and waits until Exp seconds have elapsed. Stol, Tope and Pluto provide support for this construct.

- **WaitUntilBool, *WB.*** WB receives as parameter an expression *Exp* representing a boolean value. WB stops the execution of the procedure and waits until the value of Exp becomes true to allow the execution to continue. From our set of languages, Mois and Pluto support directly this construct.

For the completeness of the example we include also a set of constructs supported by all the languages. These constructs will be used by some of the transformations we will present afterwards. *RepeatUntil* or *RU* iterates over a block of instructions, until a conditional expression yields true; *Assign* updates the value of a variable; *Add* is the mathematical addition; *Sus* is the mathematical sustraction; *Id* represents an identifier; *Gt* is the 'greater than' construct; *FuncTimeAbs2Rel* or simply *Fa2r*, is a function converting from absolute to relative time, and *FuncNow* or *Fnow*, is a function providing the current time of the system.

A quick look at the matrix gives us a first impression and some basic information that, intuitively, we expect to find in a language family.

There is a high degree of shared constructs among languages. In this case more than 70% of the constructs in the matrix are shared by all the languages.

Some of the languages will be more expressive than others. In our example, Pluto captures the functionality of all the constructs in the matrix. If we assume, just for the sake of the example, that the programs we want to translate use only this subset of constructs, Pluto could act as a universal receptor: Translators *to* Pluto should be built with the least effort. On the other hand UCL presents itself as the least expressive language. It does not consider any of the Wait constructs. A consequence could be, under the same assumption as in the Pluto case, that building translators *from* UCL should be less complex. Other properties of the language family exist. They are probably more interesting than those we have mentioned, but they are harder to see, requiring a slightly more complex analysis of the data. All the properties of the LCM, that we have considered useful, will be explained later on in Section 5.2.5, where we explain the different metrics we used.

The process of filling the LCM table with languages and constructs, indeed means annotating the grammars consistently, as shown in Section 4.2.1. For every production, according to its semantics, the user decides to which language concept it corresponds. Our system uses those annotations to index the information in the LCM, and presents it back to the user. To be

successful, this process requires that the same dictionary is used with all the languages. The user should be careful when introducing new constructs. Otherwise there is the risk of duplicate semantics, which would make the mapping overcomplicated.

The process of annotating the grammars is done manually by the user, as explained in Section 4.2. Some automated assistance can be offered in this step, by providing a recommendation of the possible categories where the production can be classified, based on the production's syntactic information. This is presented in the following section.

5.2.3.2 Assisted Constructs Classification

To classify all the constructs of a language family inside a common structure, we need to create the categories where to classify the constructs. Each category defines a set of essential semantic properties that every construct must have in order to belong to it. Constructs will be included in their corresponding category, regardless of the language where they come from.

Constructs categorisation involves two basic steps: extraction and classification.

Constructs Extraction, already explained in Section 5.2.2, consists of differentiating and extracting, as independent entities, the constructs from a language grammar. We analyse the syntactic structure of the grammar, its productions and symbols, looking for distinct semantic concepts. This analysis, that can involve a modification of the original grammar, produces as output a list of constructs. Each construct is an instance of an abstract semantic concept that corresponds to a distinct operation. These abstract concepts are the prototypical structures representing the categories in our family, and they are language independent.

Constructs Classification is the process of first, including the categories into the language family, and then linking the constructs coming from each language with their corresponding category. Each construct is an instance of some category.

There are two cases to consider for this step. First, the family is empty, and we are about to classify the first language in the family. Second, one or more languages have already been introduced inside the family, and we need to include a new language inside the family.

The first case is straightforward. The user has no practical restrictions when annotating the grammars to delimit the constructs for the first language. This first language provides the initial definition of the family: the initial set of categories and the first instance for each one of these categories. This initial definition will guide the process of mapping the other languages we will include in the family.

Even though we do not provide any evidence to support this claim, we believe that there are two alternatives providing a head start to this process:

- Use the most expressive language in terms of the variety of constructs

it considers. This provides the best mapping possibilities for the rest of the languages.

- Use the language you know better. It makes easier to detect similar features in constructs coming from other languages.

The complexity of this step lies in, first, the appropriate selection of the first language, and second, the appropriate selection of the annotations defining each category: the names of the categories need to be as meaningful and non ambiguous as possible. Our recommendation is to base the annotations on the syntactic keywords from the productions defining the construct. More on this later in this section.

The second case, when we already have some languages inside the family, implies a reduced freedom when extracting constructs from the new languages. Not only do we need to extract the constructs from the new language paying the same attention as we did before. Additionally we need to annotate the constructs according to those categories that already exist inside the family. To include in the family a construct coming from a new language, we have to check if the construct is an instance of an existent category, or if we need to create a new category. Checking whether a category already exists inside a family, involves trying to match the new construct with the prototypical structure of an existing category.

The list of categories in a family can be very long. In the worst case, the user will have to check, category by category, if the construct matches the semantics of some existing language concept. The nature of this work makes it error prone, with a high risk of missing the correct category. A quick explanation is presented next, of how we performed this analysis during our case study. Then we will present our proposal for an automated methodology that emulates the first phases of this process.

Categorisation Based on Similarities Building a language family in general, and in particular categorising the different language constructs into this family, can be facilitated if the product-line designer has a previous experience in the application domain. In practice though, this is not often possible. There is a high probability that the users from the application domain are completely unlinked from the people who are building the product-line and the translators. Two conditions are, nevertheless, required and assumed as premises.

First, the product-line designer has a solid experience and knowledge on programming languages in general. Second, the designer is at least aware of the specific application domain: a general knowledge of the domain is necessary to deal with domain-specific instructions. This specific knowledge, though, can be acquired through reference documentation, and not solely by personal experience.

These two premises are the least restrictive but at the same time important to be able to build a consistent language family. The designer needs to be

able to recognise and differentiate the different data and control structures present in a programming language and in the source code. He is required to know when an instruction can be mapped directly, and when and how it needs to be adapted through some program transformation.

When categorising the constructs from a language, two phases are traversed by the designer.

First the designer builds an intuitive opinion about the category where the construct may belong. This opinion, based on the syntax of the productions that form the construct, is derived from his experience with other languages. His previous experience with languages provides him with a *knowledge base* where he has already categorised, on an ad-hoc basis, productions and constructs from other languages. The intuitive opinion built based on this knowledge, will be confirmed or dismissed in the next phase.

The second phase is based on a more in-depth analysis of the semantics of a construct, and transforms the previous intuitive opinion into a fact, and therefore into the construct to be classified. The designer needs to use additional sources of information to be sure of the category where a construct belongs. This semantic knowledge is not included into the grammar, which contains syntactic information only. This second phase, being dependent of information from different sources, falls out of our control, and we cannot provide any support at this level, in the context of this work. The first phase, on the other hand, is completely linked to the grammar, and some assistance can be provided to the designer, though we acknowledge that the process remains highly manual.

In the next section we proceed to describe the methodology we used as product-line designers, to build the first opinion about the category where a construct belongs. We show how to express this methodology in terms of an automated constructs classifier that emulates the manual process.

5.2.3.3 Constructs Classification Methodology

The input for this part of the process is a list of constructs, from a particular language, generated according to the language concepts extraction principles presented in Section 5.2.2. In summary, the list of constructs will be iterated over, and for each construct a match with one of the existing categories in the family will be searched. If no match is found, a new category will be created.

We use an instance-based classification that is very close to the nearest neighbors techniques [78]. The languages family is described thanks to an n-dimensional space where each syntactic attribute of the constructs become one of the dimensions. Every construct and every language concepts category is described by an attributes vector, that we call its *syntactic pattern*. In the constructs, the values of the attributes are assigned according to their relevance in describing their semantics. In the categories, the values for the attributes are a combination of the values of all constructs classified in that category. For every new construct to be classified, we obtain

its candidate categories by calculating the Euclidean distance between the syntactic pattern of the construct and the syntactic pattern of each category. We hypothesise that the closer the match, the more probable to get a positive classfication. Finally, the candidate categories are analysed in-depth for semantic equivalence using the available language documentation.

Building the syntactic pattern of the constructs: a weighted list of the relevant visible attributes of the construct. Constructs are composed of one or more productions, that are composed of various symbols. The names of these symbols are syntactic keywords representing the visible attributes of the construct.

The goal of this step is to build a list with those keywords with semantic meaning, and assign each keyword a value based on their relevance to describe the semantics of the construct. Syntactic delimiters, like colons, semicolons, parentheses, and alike are discarded because of their lack of semantic meaning: they are in general used to make the code more readable. Each relevant keyword or attribute of the construct represents one dimension in the n-dimensional space of the family, and the magnitude of each dimension is the assigned value. This list of weighted attributes becomes the syntactic pattern of the construct.

The meaningful keywords differ in their importance to describe the semantics of a construct. Even though the personal criteria of the designer has an influence in this choice, the general prioritisation applied during our experiment was, starting from the most relevant keywords:

- The production's name, which in a *well designed* grammar, tends to be self-explanatory about the semantic meaning of the production. If for instance, we have some production named "IfThenElse", then we probably do not need any additional information to confirm its semantics: a classical conditional branching. Even though in general the first priority can be assigned to the production's name, the user should be aware of special cases where the production name is too generic, and provides no help for categorisation. If for instance the production has been named *Expression*, which is a very generic name, good chances are that many productions with different semantics share the name, and that more information is needed to disambiguate.

- Non-terminal keywords provide the basis for understanding a program. We use non-terminal keywords to understand the source-code of a program, and we use them as well to understand a grammar. Let us consider a typical keyword 'if', typically used for conditional control-flow structures. If we see the 'if' keyword inside a section of source code, we immediately know, based on our previous knowledge, that some kind of conditional is being used, even though other keywords are needed to decide what is the specific kind of conditional. Inside a grammar it happens exactly the same. We start the analysis by looking

for some keyword we are used to. As soon as we see a keyword that we recognise as familiar, we have an approximate idea of the semantics of the construct. We start to build an opinion based on visible syntactic similarities with our own personal categorisation.

- The names of the symbols in the left hand side of a production can be considered of a reduced utility, even though they do have some weight in our decision. The reason of the lesser importance of these keywords is because there are a few symbols like Expression, Statement, Block, etc., that are used in many productions, and therefore are too generic to effectively prune, later on, the list of candidate categories.

In Figure 5.8 we show how we extracted the syntactic pattern from "if-then-else" constructs in three different languages. The weights we used in this example –and along our validation– are normalised to a maximum value of 1, where production names are worth 100% of that value, non-terminal keywords 80% and terminal keywords 50%. For those cases where some keywords appear more than once, as you can see with "if" and "block" in PLUTO, we add weights. Finally we total the values, and normalise them to a maximum value of 1.

The list built this way becomes the syntactic pattern [72] of the construct. In the next step we use the syntactic patterns to find the best matches between them, and the candidate categories.

Calculate the syntactic pattern of the language concepts categories: a combined pattern of its instances. The goal of extracting the construct syntactic pattern, in the previous step, is to use it for classifying the construct into the adequate language concept category. Each category has its own syntactic pattern, resulting from the combination of the syntactic patterns of all the constructs already included there.

In Figure 5.9 we show how the category pattern for the IF language concept evolves as we include in it each one of the "if-then-else" constructs from Pluto, Stol and Elisa that we saw in Figure 5.8.

At first, every category receives its syntactic pattern directly from the first language included in the family. Pluto in this example. Every time a new language is included, we combine the syntactic patterns by vector addition, and normalisation again to a maximum value of 1.

Generate the ordered list of candidate categories. For every construct to be classified, we want to know the probability that it belongs to one of the categories in the family. We work under the assumption that constructs with similar semantics tend to have similar syntactic patterns, and therefore there is a higher probability that a syntactically similar construct is also semantically equivalent. Under this assumption, what we need to calculate is how close each category is from the construct. We do that by calculating

PLUTO
``if'' Expression ``then'' Block ``else'' Block ``end'' ``if'' -> If

	ProdName	NonTerminal	Terminal	Total	Weight
if	100	80 + 80		260	**1.00**
then		80		80	**0.31**
else		80		80	**0.31**
end		80		80	**0.31**
block			50 + 50	100	**0.38**
expression			50	50	**0.19**

STOL
``if'' Expression ``then'' Block Else ``endif'' -> If
``else'' Block -> Else

	ProdName	NonTerminal	Terminal	Total	Weight
if	100	80		180	**1.00**
else	100	80		180	**1.00**
then		80		80	**0.44**
endif		80		80	**0.44**
block			50 + 50	100	**0.56**
expression			50	50	**0.28**

ELISA
``if'' Expression Block ``else'' Block ``end'' ``if'' -> If

	ProdName	NonTerminal	Terminal	Total	Weight
if	100	80 + 80		260	**1.00**
else		80		80	**0.31**
end		80		80	**0.31**
block			50 + 50	100	**0.38**
expression			50	50	**0.19**

Figure 5.8: Syntactic pattern of the IfThenElse construct in different OLs.

IF

	Pluto	Pluto+Stol	Pluto+Stol+Elisa
if	**1.00**	**1.00**	**1.00**
then	**0.31**	**0.38**	**0.25**
else	**0.31**	**0.66**	**0.54**
end	**0.31**	**0.16**	**0.21**
endif	**0.00**	**0.22**	**0.15**
block	**0.38**	**0.47**	**0.44**
expression	**0.19**	**0.24**	**0.22**

Figure 5.9: *If* syntactic pattern evolution.

the Euclidean distance between the construct's syntactic pattern, and the category's syntactic pattern:

Having the construct x, represented by its syntactic pattern

$$x = \{a_0, \ldots, a_n\}$$

For every category $c \in C$, where c is also represented by its syntactic pattern

$$c = \{a'_0, \ldots, a'_n\}$$

The distance d between c and x is given by the formula

$$d(x, c) = \sqrt{\sum_{i=0}^{n} (a_i - a'_i)^2}$$

When an attribute is not present in a syntactic pattern, it is included with a value of 0. This way the syntactic patterns of constructs and categories are always aligned.

	STOL(DoWhile)
repeat	0.00
loop	0.00
until	0.00
block	0.50
expression	0.50
dowhilenot	0.00
do	0.80
exit	0.00
if	0.00
end	0.00
enddo	0.80
dowhile	1.00
while	0.80
whilenotdo	0.00

WHILE	UNTIL	REPEAT
0.00	0.00	1.00
0.00	0.00	0.22
0.00	0.00	0.44
0.19	0.31	0.41
0.19	0.31	0.41
0.00	0.00	0.32
0.31	1.00	0.50
0.00	0.50	0.25
0.00	0.50	0.25
0.31	0.00	0.25
0.00	0.00	0.00
0.00	0.00	0.00
1.00	0.00	0.00
0.00	0.63	0.00
D=1.49	D=1.81	D=1.98

Figure 5.10: Construct-category distances

The closer the syntactic patterns –the smaller the distance– the higher the probability of finding a semantic equivalence. Figure 5.10 shows an example of the *DoWhile* construct from STOL, compared against three categories: WHILE, UNTIL and REPEAT, where the constructs from the languages PLUTO, UCL and ELISA have already been classified. To the bottom of the figure we have the calculated distances. The WHILE concept distance is the closest one to the STOL construct, which in this case happens to be the correct equivalence category for the DoWhile construct being analysed.

This shallow syntactic analysis may return many hypotheses or candidate categories. A small subset of those seeming most promising –higher probability of matching– will be selected for the following in-depth semantical analysis [81].

Finding the semantically equivalent category for a construct. For a given construct, the previous step provide us with a list, ordered by increased distance, of all the language concepts in the family. We can assume based on our syntactic similarity premise, that if there exists some category functionally equivalent with the construct, it has to be near the beginning of the list. There is no rule of thumb we can use to know when to decide that there is no equivalent category. Nevertheless, it is reasonable enough to think that the user should not check more than 5 categories. Our validation experiment shows that for the cases where an equivalence existed, in 69% of the cases it was found in the first recommended category, with another 16% percent in the second recommended category. The user has to be aware though, that this process is based on an hypothetical premise. No proof exists that syntactic similarity can somehow guarantee some kind of semantic equivalence. Even more, if no syntactic similarity is found with this process, it does not mean either that no semantic equivalence exist. If a semantic equivalence is found, we include the construct in the category. Otherwise, a new category must be built.

5.2.4 Language Concepts Adaptation

In the previous sections we have shown how the mapping technique of our approach allows us to align equivalent language constructs. This mapping technique is useful for those translations where the differences between constructs are mainly syntactic. When working with families of related languages, and the family of OLs in particular, the majority of the constructs we are translating can be handled with such a mapping. However, mismatches between constructs, whenever a one-to-one equivalence cannot be established, remain present to some extent in the generated program translators. Those mismatches can be difficult to solve, and require the design of complex specific program transformations for each case.

Programming specific program transformations consumes a significant part of the effort put in building program translators. The results of this effort can be maximised if, when building a new translator, we reuse some of the transformations we have already programmed for previous translators. This reuse is possible because even if the languages in a family may be syntactically different, they have conceptually the same building blocks. If we know in advance what other language or construct could benefit from some transformation, we can try to design the transformation accordingly. Through reuse, we can reduce the "cost" of the transformations, and obtain an overall advantage.

When adding a new language to a language family, we expect that a big part of the constructs in this new language, can be directly mapped to equivalent constructs in those languages already in the family. We have already discussed this situation in the previous sections. What we also expect, and that is the subject of this section, is that among the mismatches we have pre-

viously noticed between the languages already in the family, some of them are the same in nature, as those that will be detected in the new languages. For these cases, if a transformation was designed to solve a previously encountered mismatch, there are good hopes that this transformation can be reused as is, to adapt the mismatched construct detected in the new language as well.

Program transformations are designed to solve a specific problem between some source and target languages. No guarantee exists that they will remain valid if either or both languages are replaced. This uncertainty can be reduced if we provide a mechanism to classify or index the transformations, the same way we do with the languages: transformations will be linked to a category of constructs, and therefore we assume they will work for all the constructs linked to the same category. This idea is backed up by the fact that during the design of the Language Concepts Matrix, we agree that all the constructs belonging to a certain category are functionally equivalent, regardless of the language. Transformations then can be thought as solving a category of problems.

Our approach for reusing transformations proposes using two interrelated matrixes to index the languages, constructs, and transformations, inside a specific language family. We can use the information in those tables to derive the appropriate relations and paths between unmatched constructs and existing transformations. The Language Constructs Matrix, LCM, where we relate languages and constructs, was already explained in Section 5.2.3.1. In the following section we show how to relate constructs and transformations, and we provide an example of how we use this information to reuse existing transformations.

5.2.4.1 Indexing Transformations: the Transformation Constructs Matrix, *TCM*

The Transformation Constructs Matrix, TCM, is a two dimensional matrix where the relationships between language concepts and transformations acting on the constructs linked to those language concepts are shown.

Let us first review a simple transformation, its basic structure, and how it is used. A transformation is a rewriting expression composed of three parts: a name, a matching pattern, and a replacement pattern.

```
(name)
match
==>
replace
```

The name is a unique id allowing us to identify the transformation unequivocally. The name does not have any influence on the execution of the transformation, but it allows us to reference it precisely. The matching pattern presents the construct that will be transformed. The replacement pattern shows the combination of constructs that will be written in place

of the matched construct. Some additional data and operations can be necessary for building the replacement pattern, like conditions, but we do not include them here, because they tend to be implementation-dependent.

In a rewriting transformation the *replace* side is functionally equivalent to the *match* side. This is guaranteed by the designer of the transformation. What we need to know, when trying to reuse a transformation, is if all the individual components of the replace side will be supported by the *new* target language. If the target language does support all the elements in the replacement side of the transformation, we consider the adaptation complete and equivalent. When one or more of the elements in the replacement pattern are not supported by the new language, we need to recursively search for another transformation capable to adapt these unsupported elements. If we cannot find a chain of transformations providing only constructs supported by the new language, then the we have a partial adaptation, and we will need to design additional transformations.

The Transformations-Constructs-Matrix, TCM, was designed to help the designer of the languages family, in discovering how to combine existing transformations, and when to design additional transformations.

The TCM table is very similar to the LCM table. It has one column for each transformation, and one row for each construct in the family. The columns are indexed according to the *match* construct of the transformation, which finally implies that we are relating constructs against constructs. The cells or row-column intersection of the TCM admit two values:

- An *I* means that the construct is modified, or required by the transformation.
- An *X* is shown when the transformation produces among its result the corresponding construct.

Along this section we will come back to the *WAIT* example introduced in Section 5.2.3.1.

	WA2WR	WR2WB	WB2RU	WA2RU	WR2RU
WaitUntilTimeAbs, WA	I			I	
WaitForTimeRel, WR	X	I			I
WaitUntilBool, WB		X	I		
RepeatUntil, RU			X	X	X
Assign		X			X
Add		X			X
Sus	X				
Id		X			X
Gt		X		X	X
FuncTimeAbs2Rel, Fa2r	X			X	
FuncNow, Fnow	X	X		X	X

Figure 5.11: The working example TCM table.

Figure 5.11 presents one of the TCMs we will use for our examples. There are five transformations we have indexed there, whose names are shown as

headers of the columns. For instance the first transformation to the left is named WA2WR. Its match construct, the one that will be transformed, is WaitUntilTimeAbs, or WA for short. In the first column of the table we find as row headers the names (and shortnames) of all the constructs present in either the match or the replacement patterns. If we analyse again the first transformation, WA2WR, we can see that it produces in its replacement pattern the constructs WR (short name), Sus, Fa2R and Fnow. The WA2WR transformation is presented in Figure 5.12.

Designing a transformation that adapts some unmatched construct, provides to the target languages an indirect support for that construct, which is functionally equivalent to a direct mapping. If we follow the example from Figure 5.7, we can see that the construct WA is not supported by the language Tope. If we want to translate programs from Stol to Tope, we need to build a transformation adapting WA into something Tope understands. If we can come up with a transformation like WA2RU, shown in Figure 5.14 then we have provided Tope with an indirect support for the construct WA. Analysing the replacement pattern of the transformation adapting WA, we can see also a very interesting circumstance. The replacement constructs produced by the transformation are in the intersection of all the languages. We have, therefore, an adaptation useful not just for Tope but also for Ucl and Mois. It is not always easy or possible to come up with something as generic as WA2RU. In many cases we can only produce transformations like WA2WR, in Figure 5.12, which is a valid adaptation only for Tope.

The last step, once the TCM has been filled with the information from the designed transformations, is to put them together in the translator. Several strategies to apply the transformation rules exist. Some of these strategies, like in the case of ASF+SDF, are automatic. The rewriting system takes the required decisions. Some other strategies are more user-defined, like in the case of Stratego, where the user can provide the order. The many different kins of strategies are thoroughly explained in Visser's survey [125]. In our methodology, we give preference to a user-defined strategy, where the order of the sequence of rules can be chosen for every translator. Thanks to the TCM, we can provide to the user a suggested order in which the rules can be applied. The technique defines a partial order between the transformations, and then apply a topological sort. The partial order rule we use is the following:

having :

$t_1, t_2 \in Transformations,$
$c \in Concepts$

then:

$if\ t_1(c) = I \wedge t_2(c) = X$
$\Rightarrow\ t_1 < t_2$

For every two transformations t_1 and t_2 using the same concept c, if t_1 requires c (marked with an I in the TCM), and t_2 generates c (marked with and

X in the TCM), then t_1 has a lower precedence over t_2, and therefore t_2 has to be applied before t_1 (higher precedence executes before lower precedence). In the example from Figure 5.11 the partial order set is:

$$WR2WB < WA2WR, WB2RU < WR2WB, WR2RU < WA2WR$$

Once the partial order has been defined, a topological sort provides the suggested order. For those cases where cycles are detected: $t_1 < t_2 \wedge t_2 < t_1$, the user is alerted about the rules involved in the cycle. The user, then, can adapt the solution choosing the more convenient order for every translator.

5.2.4.2 An Adaptation Example.

This example emulates, on a reduced scale, the full process of building the program translators, once we have already categorised all the constructs into the LCM table. We have divided the example in four steps. In each step one new translator is built. During the first three steps we set up the example by adding the languages one by one, and by adapting their unmatched constructs through a series of transformations. In the fourth step we show how the existing transformations are reused to build a new translator.

1) The first translator we want to build is the *Stol* $\rightarrow$ *Tope* translator. In the corresponding LCM table 5.7 we can see that a mismatch exists in the construct `WaitUntilTimeAbs`. This construct exists in Stol, but it is not natively supported in Tope. Therefore we need to build a transformation to adapt this construct. Analysing the semantics of the construct, we come up with a functionally equivalent transformation using the Tope construct `WaitForTimeRel`, plus some other auxiliary constructs. The designed transformation is presented in Figure 5.12.

```
WA2WR)
WaitUntilTimeAbs($texp_abs)
==>
WaitForTimeRel(Sus(Fa2r($texp_abs), Fnow()))
```

Figure 5.12: The WA2WR transformation.

What the WA2WR transformation of Figure 5.12 does is obtaining the difference between the absolute time, and the current time, and then passing this value into the `WaitForTimeRel` construct. If we think in terms of a concrete source code example, the WA2WR transformation would look as in Figure 5.13.

The WA2WR transformation of Figure 5.12 works perfectly to fulfill our current goal, and is probably the best alternative if we want to remain as close as possible, both syntactically and semantically, to the original intention of

```
wait until 2009-10-01:23:15:30
==>
wait ( clock scan 2009-10-01:23:15:30 - clock seconds)
```

Figure 5.13: A concrete example with the WA2WR transformation.

the code. There are however other ways to transform this construct, like for instance the transformation WA2RU, shown in Figure 5.14, whose concrete source code example is shown in Figure 5.15.

```
WA2RU)
WaitUntilTimeAbs($texp_abs)
==>
RepeatUntil(
  Gt(Fnow(), Fa2r($texp_abs)),[]
)
```

Figure 5.14: The WA2RU transformation.

```
wait until 2009-10-01:23:15:30
==>
repeat { } until (clock scan 2009-10-01:23:15:30 > clock seconds)
```

Figure 5.15: A concrete example with the WA2RU transformation.

The advantage of WA2RU (Figure 5.14) is that it adapts the construct not only for the Tope language, but also for the Ucl and Mois languages without the need of additional transformations. The disadvantage is that the resulting code, has less in common with the original code, than if we use the WA2WR alternative.

There is no good or bad transformation in this case. Both have advantages and disadvantages, and both fulfill the purpose. For the sake of the example, let us choose the first alternative, WA2WR.

2) Now we want to build the translator $Tope \rightarrow Mois$. Looking at the LCM table 5.7, the mismatch is found in the `WaitForTimeRel` construct, supported by Tope, but non existing in Mois. We need again to design a transformation to adapt this construct, and we decide to build the WR2WB transformation, shown in Figure 5.16:

The main construct produced by WR2WB is `WaitUntilBool`, that we consider as the semantically closest construct in Mois. As in the previous step, there are other possible alternatives for this transformation that we are

```
WR2WB)
WaitForTimeRel($texp_rel) //{[$t := _newId()]}
==>
Assign(Id($t), Add(FuncNow(), $texp_rel))
+
WaitUntilBool(Gt(FuncNow(), Id($t)))
```

Figure 5.16: The WR2WB Transformation.

not considering.

3) The last step to complete the setup of our example is to build the translator $Mois \rightarrow Ucl$. The mismatch is located in the `WaitUntilBool` construct, not present in Ucl (indeed Ucl seems to be be the weakest language regarding the WAIT family of constructs). The transformation we choose to implement is WB2RU, in Figure 5.17:

```
WB2RU)
WaitUntilBool($b)
==>
RepeatUntil($b,[])
```

Figure 5.17: The WB2RU Transformation.

Now the setup for our example is complete, and in the next step, *step 4*, we illustrate our methodology for transformation reuse.

4) Suppose that the translator we need to build now is Stol → Ucl. We already have a database of transformations we have generated for other translators, and we would like to know what can be reused. The base condition to do this is to fill the LCM and TCM tables, as shown in figures 5.7 and 5.11 respectively. Next we can initiate the following process:

1. As in the previous steps, if we scan the LCM table top-down, we find the first mismatch between Stol and Ucl in the `WaitUntilAbsolute` construct, present in Stol, but not supported by Ucl.

2. We jump now to the TCM table in Figure 5.11 and check, in the *match* index, if there is some transformation already designed to adapt the `WaitUntilAbsolute` construct. We find the WA2WR transformation in the first column of the table.

3. Next, while still in the TCM table, we check in the WA2WR column if all the symbols produced by the transformation are supported by Ucl. We find that there is one construct that is not supported: the `WaitForTimeRel` construct.

4. We repeat the second step, but now looking for a transformation adapting the `WaitForTimeRel` construct. We find the WR2WB transformation in the second column of the TCM.

5. We repeat the third step with the WR2WB transformation, and we find that the `WaitUntilBool` construct is not supported by Ucl.

6. We repeat the second step once more, and we search for a transformation adapting the `WaitUntilBool` construct. We find the WB2RU transformation in the third column of the TCM table.

7. Repeating the third step, we find now that all the constructs produced by the WB2RU transformation are supported by Ucl. This makes our complete search successful, and we can emit as result the sequence of transformations:

$$WA2WR \Rightarrow WR2WB \Rightarrow WB2RU$$

The LCM table gets updated as shown in the excerpt in Figure 5.18.

	Stol	Tope	Ucl	Mois
WaitUntilTimeAbs, WA	X		WA2WR	
WaitForTimeRel, WR	X	X	WR2WB	
WaitUntilBool, WB			WB2RU	X
RepeatUntil, RU	X	X	X	X

Figure 5.18: The LCM table updated.

As additional positive side effects of this example, is worth noticing the following:

- We have found a solution not only for the mismatch we noticed with the `WaitUntilAbsolute` construct, but also for the second mismatch between Stol and Ucl: the `WaitForRelative` construct. Analysing the steps from the preceding procedure backwards, we can see that once we find the WB2RU transformation, it makes the transformation WR2WB fully compatible with Ucl, and this transformation also provides an adaptation for the `WaitForRelative` construct.

- Even if for the $Stol \rightarrow Ucl$ translator, it is not necessary to provide a compatibility with the `WaitUntilBoolean` construct, because it is not included in any of the two languages, the WB2RU transformation provides this compatibility anyway, therefore if at some time we need to build for instance the translator $Mois \rightarrow Ucl$, we already have found an adaptation for the mismatch with the `WaitUntilBoolean` construct.

- Moreover, for the current example, Ucl becomes fully compatible with all the four languages in the set, because we have found adaptations for all the constructs that Ucl was not supporting.

- Finally, If we repeat this procedure with all the languages in the set, we will see that all the mismatches are solved with only those three transformations, and we can build any full translator in that family without designing additional transformations.

5.2.5 Evaluating the Language Family. Metrics and Properties

A language family is well described by the LCM and TCM tables, whose visual representation already give us a good overview of some of the global properties of the family. Nevertheless, languages are big structures, with many constructs, and we need more efficient ways of describing the state of our product-line, and the way it evolves as we keep adapting the mismatches we find.

In this section we present some basic properties and metrics that can help to assess different perspectives of a language family. These metrics can be used to get a better feeling of how the adaptability between languages increases (or decreases) as we keep designing and reusing transformations.

Our proposal of metrics turns around the concept of *entropy*, which in short is a measure of the disorder of a system. The concept of entropy comes originally from thermodynamics, but it has been adopted by many branches of computer science. In information theory the Shannon entropy [103] is used to measure the undeterminacy of a message, which has direct implications on knowing its expected compression rate. In the field of biology Rao [98], and this is key to our metrics proposal, measures diversity and similarity among populations thanks to the quadratic entropy that incorporates the functional differences among species. Machine learning techniques make extensive use of entropy, for instance to determine how well some attribute classifies the training data [78]. Calera-Rubio et al. [21] calculate the relative entropy between regular tree languages to measure the similarity of grammatical inference learning methods. Closer to software engineering and to the study of programming languages, Roca [99] uses entropy to measure the structural complexity of software, in an effort to determine the safetyness of critical redundant software. Krein et al. [61] propose language entropy as a metric to characterise how, in a multi-language development environment, authors distribute efforts between those languages, and how that distribution affects productivity.

We use the general principle of entropy to determine how close, and therefore how compatible, the languages in a family are, in terms of their language concepts. We start by defining the basic compatibility measures step by step, to finish with the overall family entropy metric. Then we derive some auxiliary metrics that will help us find out how our efforts to adapt mismatches influence the global compatibility.

Language To Language Distance, $L2LD_{ij}$. Considering that in our approach, languages are characterised by the language concepts they use, which

are categorical attributes, the more appropriate measure of the distance between two languages is Podani's simple matching coefficiency [93], shown in Figure 5.19. The use of Podani's distance was already proposed by Botta-Dukat [18] when dealing with nominal, unordered categorical, functional attributes.

$$L2LD_{ij} = \frac{u_{ij}}{|C|}$$

Figure 5.19: Distance between two languages, $L2LD_{ij}$

In this formula i and j represent two languages in L, which is the set of languages in the family. C is the set of language concepts in the family which results from the union of the concepts provided by all the languages in L: $C = \bigcup_{l \in L} C_l$. Finally, u_{ij} is the number of concepts whose value in the LCM table is not the same in languages i and j ($LCM_{[c,i]} \neq LCM_{[c,j]}$).

The C set of concepts must remain constant during all calculations inside the family, to avoid bias. Therefore even for this language to language case, we always use the full set C. For cases where you are interested exclusively in the distance between two languages, and no further comparisons will be made against other languages, it can be more appropriate to use only the subset $C_i \cup C_j$

An L2LD = 0 means there are no differences between the two languages: they use the same set of concepts. An L2LD = $|C_i \cup C_j|/|C|$, means a complete absence of common traits. The highest value of L2LD tends to 1.

Average Language Distance,LAD_l. The next step –and metric– calculates de mean distance between one language and the other languages in the family, and it is shown in Figure 5.20.

$$LAD_l = \frac{\sum_{i \neq l}^{L} L2LD_{li}}{|L| - 1}$$

Figure 5.20: Language Average Distance, LAD_l

This value is simply the addition of the individual distances between l and each of the other languages, weighed by the number of distances calculated. If in doubt about how well some language belongs to a

family this is the metric we have to use. LAD values start from 0, when the language is a perfect representative of the family, and tend to 1 when it has nothing in common with the rest of languages.

Languages Compatibility Index, *LCI*. This is probably the most relevant metric. It gives us a measure of how compatible, overall, the languages in a family are.

$$LCI_L = \frac{\sum_{i=1}^{L-1} \sum_{j=i+1}^{L} L2LD_{ij}}{1/2(|L|(|L|-1))}$$

Figure 5.21: Languages Compatibility Index, LCI

The formula we use, shown in Figure 5.21, is derived from Ganeshaiah's avalanche index [45, 44] and average taxonomic diversity [46]. In turn those metrics adapt Rao's quadratic entropy [98].

The formula adds the distance between every couple of languages in the family, and weighs it for the total number of unique combinations.

In the ideal case, having a family where all the languages are compatible with all the concepts, would produce an LCI of zero.

	Stol	Tope	Ucl	Mois	Pluto
Stol	0	0.09	0.18	0.27	0.09
Tope	0.09	0	0.09	0.18	0.18
Ucl	0.18	0.09	0	0.09	0.27
Mois	0.27	0.18	0.09	0	0.18
Pluto	0.09	0.18	0.27	0.18	0

LAD
0.16
0.14
0.16
0.18
0.18

LCI	0.16

Figure 5.22: Distances sample between OLs.

In Figure 5.22 we can see an example of these metrics, based on the LCM table from Figure 5.7. Inside the table, each cell contains the L2LD values between pairs of languages. The rightmost column presents the LAD for each language, and at the bottom of the table we can see the LCI general value for this sample of the operations languages family.

This sample considers 5 languages, and a set of 11 concepts. From this set of concepts three of them, those related with the WAIT functionality, are not compatible with all languages. It is this group of three concepts that contributes negatively to the entropy of the table.

If we fill the table with the three transformations developed in the example from Section 5.2.4.2, the LCI becomes 0.

Direct Language Compatibility Index, *LDCI*. The LCI is very general, and can be used at any moment we want to know more about the overall compatibility of our family, for instance after introducing some set of transformations. The Languages Direct Compatibility Index is a slightly restricted form of LCI.

This metric is the same as the LCI, but with the restriction that it only considers the concepts provided natively by the languages. In other words, this metric does not consider any adaptation introduced by the transformations we include in the system. Once we categorise all the languages in the family, its value remains constant.

We use the LDCI as the representation of the initial state of the languages family. Variations from this metric will give us the measure of how the system has progressed in terms of compatibility.

This set of metrics complements the methodology to build a language family, with the ability to first, assess the current state of the family, regarding language diversity, and then measure its evolution in terms of compatibility. Chapter 6 will extend on the use of these metrics.

The next section presents the final step in building a language family, namely the final generation of the required translators, based on the structures built so far.

5.3 Generating and Testing

In the previous phase, structuring, we put together all the languages and language concepts in a categorised structure, the LCM. Thanks to the LCM, it is easier to notice mismatches and incompatibilities between languages. The mismatches were solved thanks to program transformations implemented by the language family designer. These transformations were put together and related with the constructs they act upon, in another structure similar to the LCM, the TCM By using both the TCM and LCM together, we showed how it was possible to reuse existing program transformations to solve additional mismatches. Finally, and unless there are unsolvable mismatches (which is an open possibility), we can obtain full compatibility –direct or adapted– between the languages that we need to translate. At this point, generating a translator becomes an automatic process, based on what we already fed into the LCM and TCM structures.

The basic structure of a translator is depicted in Figure 5.23. The original program is parsed thanks to the source language specification, and an abstract syntax tree conforming to the languages family structure is produced. The AST is sent to a rewriting engine, where the program transformations

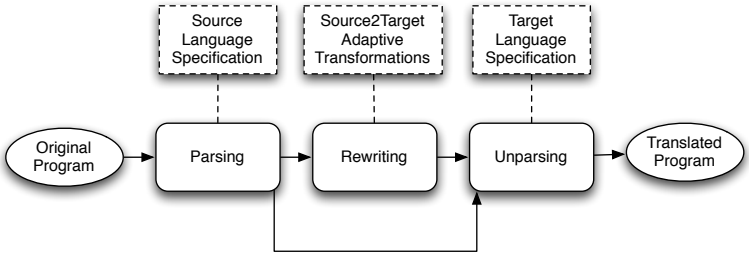


Figure 5.23: Translator structure.

to adapt the specific compatibilities between the two languages are applied. Finally, thanks to the target language specification the program is unparsed, completing the process. For those rare cases where no mismatches between source and target languages where detected during the structuring process, then the translator sends the AST directly to unparsing. No adaptive transformation is required.

As mentioned before, there is always a possibility that for some mismatches between languages, no combination of program transformations can be found to solve the conflict. In that case the translator will be incomplete, and the untranslated pieces of code will be marked with special comments. Of course, thanks to the LCM and TCM tables, the designer knows beforehand where those hard mismatches exist, and what to expect from the translation.

Finally, there is the non mandatory, though highly recommended phase of Testing. The testing phase does not differ from the language to language technique we explained in Section 4.5. It consists of verifying the observation equivalence of the original and translated programs.

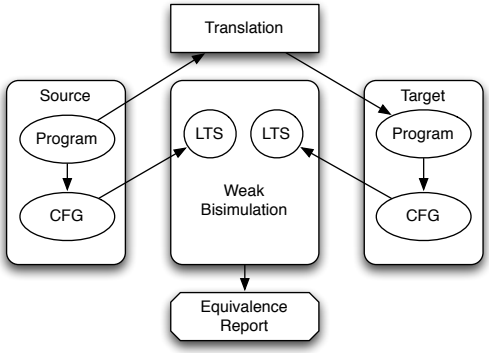


Figure 5.24: Equivalence verification.

In the schema of Figure 5.24, we summarise the main lines of the process. Once the program is translated, we generate from both source and target

programs, their respective control-flow graphs, based on their control-flow semantics, included in their language specifications. Then, we run a weak bisimulation process with the labelled transition systems generated from both control-flow graphs, and test if the equivalence holds. In the case of problems during the bisimulation, markers will be included in the graphs and in the source code of source and target programs. These markers will help the designer to detect the exact nature of the problem, go back to the structuring phase, correct the transformations, and regenerate the translators.

5.4 Conclusions

The main goal and contribution of this chapter was to provide a methodology to build families of language translators, based on two approaches:

- translation-wise, we used the language to language approach shown in Chapter 4.
- structure-wise, we adopted the product-line approach introduced in this chapter.

Along this chapter we presented first the product-line model adapted to our specific case of programming language families. Then we showed how to build a product-line, by organising the process in four well defined phases: scope definition, structuring, translator generation and testing.

Additional contributions of the chapter are:

- A set of guidelines to prepare a grammar, independently from other grammars. That way, constructs and language concepts can be easily extracted and categorised into the family structure.
- A methodology for language concept categorisation and program transformation reuse, based on a double matrix structure: the language concepts matrix, LCM, and the transformation constructs matrix, TCM. Thanks to this structure we are able to evaluate the current state of a family, and take better decisions regarding mismatches adaptation, with a reduced number of program transformations.
- A tool that provides automated assistance for language concepts categorisation, based on the syntactic similarity of constructs.
- A set of metrics to describe a language family and evaluate its evolution in terms of the compatibility of its members.

The methodology proposed in this chapter applies in cases where we need too build many different translators, from a group of syntactically similar programming languages. It improves the understanding of the languages involved, and helps reducing the number of program transformations that need to be implemented, thanks to its global structure.

It does not, however, deal with specific problems related with (functional) language incompatibilities and program transformation design. Therefore translation for certain languages remains incomplete.

In the next chapter, we exemplify and validate this methodology in detail on the specific family of Operations Languages.

6 Validation

Along this thesis we have explained our approach to build families of language translators. The approach applies a methodology based on a combination of different techniques: a product-line approach providing the support for a reusable translator framework; a grammar convergence reverse-engineering approach enabling the extraction of common features from programming languages and programs, and a language-parametric grammarware approach providing the specific translation and transformation techniques.

The purpose of this work has not been to propose yet another technique for program translation. We rather tried to use as much as possible state-of-the-art existing techniques, to build a framework that can generate a large number of program translators, among languages belonging to a same family, with an effective reduction of the programming effort involved. In this chapter we validate our approach, by building a set of translators for the family of operations languages. The translations are tested on a group of procedures defining some typical operations in a space mission.

6.1 Preliminary Case Studies

Before presenting the validation of the product-line approach to program translators, we first introduce some preliminary studies, mainly related to language-to-language translation. These studies already give indications of the power of the annotated grammars technique described in Chapter 4, which is the basis of our approach.

6.1.1 The IRL Case Study

The MOIS system and language are successfully used in the space operations environment to design operations procedures for space missions [95, 96]. One of the goals of the APPAREIL project [84] was to study how MOIS could benefit from a generic mechanism to import existing procedures programmed in different operations languages.

The traditional way to import such procedures, used so far by MOIS engineers, was to use attributed grammars and ANTLR parsers to build the required importers. After a few translators were built with this approach, it became clear to them that this approach was highly time-consuming, both when developing and especially when maintaining the importers.

After further analysis in collaboration with the MOIS team, and thanks to

the support of the CWI's SEN1 Laboratory[1], we became convinced that the special characteristics of an environment like ASF+SDF [118] could greatly improve the process of building such program importers. Therefore, using ASF+SDF, a preliminary experiment was performed, where we *manually* built a number of translators between PLUTO [42], UCL [7] and a reduced version of MOIS, that we will call IRL (IRL stands for Intermediate Representation Language).

We started with a subset of constructs for these languages, consisting mainly of control-flow structures, which is what all operations languages have in common. We manually created four translators: PLUTO to IRL, IRL to PLUTO, UCL to IRL, and IRL to UCL.

The total number of ASF+SDF rewriting rules we had to implement for these four translators was 91, but the implementation of 73 of these rules (about 80%) followed a repetitive pattern. The rewriting rules served as a kind of mapping between source and target grammars, with an almost one-to-one correspondence between productions and non-terminals. Only 18 of all the rules (slightly less than 20%) were less trivial, requiring more knowledge than what could be deduced from the grammar.

While conducting this experiment, we thus experienced a high-level of repetition. In addition, declaring the sometimes complex mappings between language concepts required high technical skills. As such, this initial experiment motivated and justified the need for a more automated approach, that could generate automatically a significant part of the rewriting rules.

6.1.2 The Stol-to-Mois Case Study

Based on the data gathered from the previous experiment, where we built program translators completely by hand, we developed a first prototype of our APPAREIL tool as described in Chapter 4. Then, we used that tool to generate a full translator from Stol to Mois. The goal of this experiment was not only to build a proof-of-concept prototype, but also to test its validity and usefulness for one of the clients (Panamsat) of our industrial partner (RHEA Systems, the creators and owners of the MOIS system). A final version of the translator that was generated by the APPAREIL tool was sent to that client, and we have been informed by our industrial partner that it was successfully used to import Stol procedures into their Mois tool.

The development time of this prototype, from the moment we started preparing the grammars, until we generated the first version ready to be tested with complete Stol procedures, was about two months. The project was in our hands (one full time researcher) with the support of one person from the industrial partner's technical team (about twenty percent of his time).[2] An interesting comparison, extracted from conversations with the

[1] http://www.cwi.nl/en/research-groups/Software-Analysis-and-Transformation

[2] After this point many delays, mostly unrelated to the development of the translator, made it difficult to track the total time invested exclusively in the project.

industrial partner, indicates that in a very similar project where a translator from Elisa to Ucl was built, one full time programmer invested thrice the time working completely on ANTLR, until he reached the same testing stage as we did. Unfortunately, we have no knowledge about other variables that could have influenced this significant difference in development time.

Regarding the size of the code of the generated translator, two separate parts should be distinguished: the code generated automatically by the APPAREIL tool based on the grammar specifications, and the code built by hand to cope with the mismatches that could not be addressed automatically by the tool. The code generated automatically counted 3.775 lines in 376 ASF+SDF rewriting rules. This code was generated from the specifications (annotations) that were added to the grammar files by hand: 229 lines of code. The manually written part of the code was programmed in Java, counting 1.182 lines distributed over 58 methods.

If we compare this data with the data obtained from the previous experiment, we can see that it goes in the direction of what we expected to achieve: in number of lines of code we were able to generate 63% of the code automatically; in number of functions we generated 85% of the total code by automatic means.

6.2 The STOL Validation

This section can be considered as an instantiation of the product-line approach explained in Chapter 5. The activities performed during the experiment reflect the organization suggested in that chapter.

In the following sections we describe, first, the set-up of the experiment and why the case was chosen; second, the methodology that was followed, which is mainly a quick summary of the approach; third the experiment itself, where we describe the problems we encountered, how we solved them, and the overall results; and finally we discuss the results obtained, providing some interpretations, advantages and disadvantages of the approach, and general conclusions.

We were first confronted with the problem of program translation in the context of a research project with a company, RHEA Systems, specialised in the domain of space-mission operations planning. The company has a software suite, MOIS [3], which amongst others, provides tool support for designing space operations procedures. Considering the sheer amount of existing procedures, designed in one of the many existing operations languages, it was interesting and necessary to study different alternatives to build program translators between these languages.

All the information we received for the experiment, and the project in general, was provided by the company. The strong constraints regarding confidentiality, very common in that domain, restricted the flow of informa-

[3] Manufacturing and Operations Information System

tion towards us. This restriction is manifested especially in the small number of test procedures we received.

The company provided us with documentation on six different operations languages: Stol, Mois, Pluto, Ucl, Tope and ELisa. We received, for the testing purposes, 10 Stol procedures that had been obfuscated previously to hide the real names of some elements, especially telemetries and commands. Nevertheless, the company certified these procedures as being good and faithful representatives of what is generally found in the domain. These procedures use a set of directives from a specific mission running on an EPOCH[4] control system.

The experiment basically consisted of building a product-line with this family of six operations languages. We then generated a set of five translators from Stol to the rest of the languages in the family. Finally, the translators were tested on the group of 10 Stol procedures.

6.3 Methodology

The methodology we used for the experiment is explained in detail in Chapter 5, and summarised in Figure 6.1. There are three main cyclic steps we follow until we reach the final product.

In the first step, we align the grammars. To differentiate the constructs, the grammars are analysed and modified when necessary, applying the recommended rules of thumb from Section 5.2.2. The constructs are then classified in categories, based on their syntactic patterns, as shown on Section 5.2.3.3. This way we obtain a first initial version of the LCM. A manual revision of the LCM table is then performed, to look for imprecisions in the categorisation process. From this part of the process we can evaluate how our actions affect the evolution of the family, thanks to the set of metrics defined in Section 5.2.5.

The second cycle is for the adaptation of the mismatched constructs. We use the LCM and TCM tables, as presented in sections 5.2.3.1 and 5.2.4.1, to decide what are the mismatches we should try to solve first, to better reuse the transformations we need to program. The LCM and TCM tables are updated with the information from the transformations we build. The cycle continues until we have solved, if possible, all the mismatches between language constructs, for the translators that need to be generated.

Finally, in the third cycle, we generate the translators and translate the set of test procedures (Section 5.3). We verify the results using the verification tool, and when necessary we go back to the previous adaptation cycle, to correct the inaccurate transformations. We regenerate the translators, and retest.

Along the entire process the measures obtained thanks to the metrics provided, are used to assess whether the decisions we have taken so far, are

[4] http://www.integ.com/

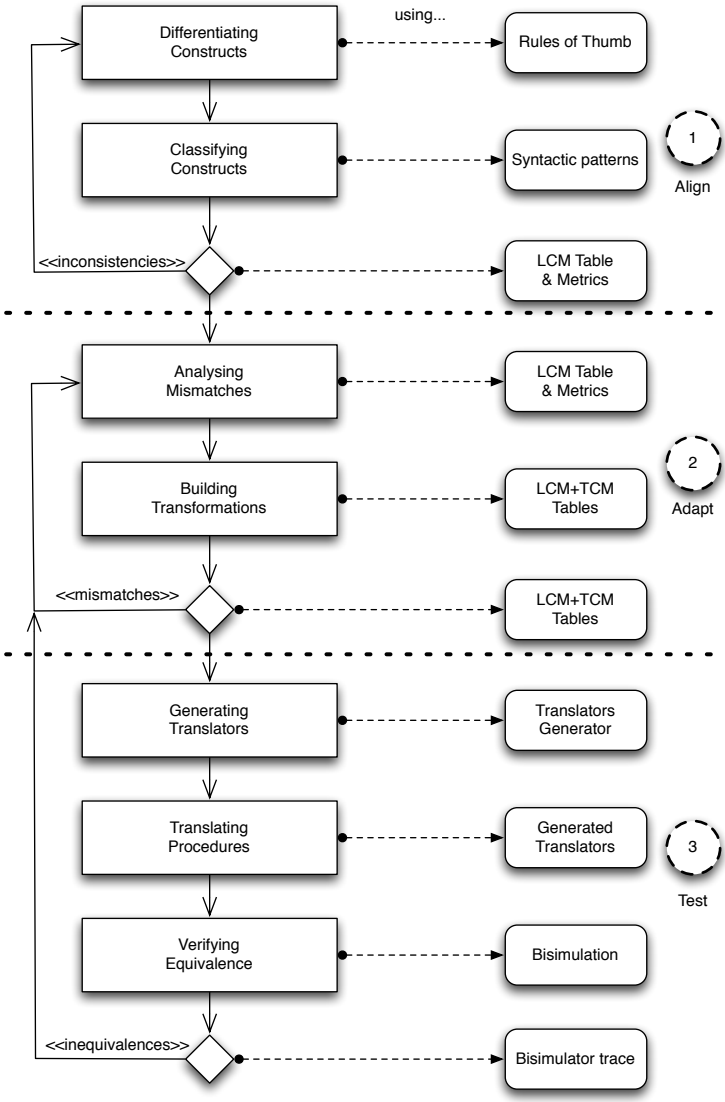


Figure 6.1: Methodology Summary

d(x, y)	Break-Statement
Break-Statement	0.8
Exit-Statement	1.05
ArithmeticExp	1.62
Continue-Statement	1.62
Function	1.65
Event-Declaration	1.65
Equals	1.65
Return-Statement	1.66
Timeout	1.67
BoolNot	1.69
Expression	1.69

Figure 6.2: Concept-Construct Syntactic Distances

giving the expected results or not.

6.4 The Experiment

Differentiating the constructs. Our main concern in this part of the process was to convert all the grammars to a GLR style. We focused on redesigning those productions that in their original version were in a LL style. This is often the case for productions dealing with precedence and associativity issues. The grammars from Pluto, Ucl, Tope and Elisa were suffering from these kind of problems, especially in the productions dealing with arithmetic and boolean expressions.

The rules of thumb recommended in Section 5.2.2, for aligning the grammars were not applied exhaustively. We wanted to keep some "raw" material to better test the assisted constructs classification technique. We did, however, get rid of some unnecessary redundant symbols, and modified the names of some other symbols to facilitate the task of extracting the syntactic pattern of the constructs in the next step. The renaming was mainly to allow an automated word splitting. For instance, the *DoWhile* construct was renamed into *Do-While*, introducing a dash between the two words. Only a few of these cases were considered necessary.

Classifying the constructs. With the grammars ready to be aligned, we first decided on the order in which the languages will be processed. The first language we include in the LCM is specially important, because it provides the initial language concepts definition for the LCM. We used Stol as first language, because it was already a well known language for us. An initial prototype for translating Stol programs to Mois had been made before, and it was tested and used by the company. Stol's grammars could be considered as a solid initial set of LCM categories. The rest of the languages were classified under a similar criteria: how familiar we are with their grammars and concepts. The final order was: Stol, Mois, Pluto, Ucl, Tope and Elisa.

(Match Position)	Mois	Pluto	Ucl	Tope	Elisa	TOTAL	%
1	29	92	89	61	120	391	69%
2	74	4	3	9	2	92	16%
3	2	1	3	4	1	11	2%
4		1	2	3	1	7	1%
5	1	1	2			4	1%
6	1		1			2	0%
10		1				1	0%
14			1			1	0%
16	1					1	0%
18		1				1	0%
19				13		13	2%
20	2			22		24	4%
25			1			1	0%
27		1				1	0%
28				1		1	0%
30				1		1	0%
31		1				1	0%
53	3					3	1%
55			1			1	0%
65			1			1	0%
84				1		1	0%
85				1		1	0%
87				1		1	0%
89				1		1	0%
95				1		1	0%
105	1	1				2	0%
122		1				1	0%
130					1	1	0%
133				1		1	0%
141			1			1	0%
147					1	1	0%
TOTAL	114	105	105	120	126	570	

Figure 6.3: Syntactic Classification Assistant Results

Each language was classified on top of the initial set of categories provided by Stol, using the recommendations provided by the syntactic classifier assistant. A small example of how these recommendations are presented is shown in Figure 6.2. The categories inside the LCM are shown to the left, and the constructs provided by the language are shown at the top of the table. The table shows the first 10 suggestions presented when classifying Elisa's *Break-Statement* construct.

The results of using the classification assistant are presented in Figure 6.3.

The first column shows the position where a positive recommendation was found. The cells of the table show the number of constructs whose correct category was recommended in that position, for each language (Stol is not shown because being the first language, it was directly included in the table). For instance in the case of MOIS, for 29 of its 114 constructs the recommendation shown as the first alternative was correct, and for 74 constructs the correct alternative was recommended in second position. The column at the right of the figure shows in average the percentage of positive recommendations for each position were a recommendation was found. In

(L2LD)	Stol	Mois	Pluto	Ucl	Tope	Elisa	LAD
Stol	0	0.12	0.25	0.24	0.14	0.15	0.18
Mois	0.12	0	0.3	0.28	0.22	0.23	0.23
Pluto	0.25	0.3	0	0.19	0.24	0.22	0.24
Ucl	0.24	0.28	0.19	0	0.13	0.17	0.20
Tope	0.14	0.22	0.24	0.13	0	0.08	0.16
Elisa	0.15	0.23	0.22	0.17	0.08	0	0.17
						LCI=	0.20

Figure 6.4: Initial Similarity Metrics

LCM	Stol	Mois	Pluto	Ucl	Tope	Elisa
ArgsDeclaration	X	X				
Arguments			X	X		
Parameter	X	X		X	X	X
Parameters	X				X	X
Command-Args	X	X				
Function-Args	X	X				

Figure 6.5: Some "Arguments" Inconsistencies

69% of the cases the first recommendation provided by the assistant was correct: it corresponded with our own manual classification based on the documentation. In another 16% of the cases the correct category was found in the second recommendation. For the other 15% of the cases, either we did not find any positive recommendation, or the recommendation was buried too deep in the list as to be of any practical use: if we do not find a positive in the first 5 positions, it is the better to postpone that construct till the end, when we have significantly reduced the list of available categories. All languages presented a similar behaviour.

Once we classified all the languages in the LCM, we applied the compatibility metrics on this first version of the LCM table. Figure 6.4 show these results. The cells on the language-to-language intersection present the Language to Language Distance, L2LD (the values to the left of the diagonal mirror the values to the right). The column to the right-side of the table shows the Language Average Distance, LAD, and finally, the last row of the table shows the Languages Compatibility Index, LCI, of the family.

Second revision of the LCM After the first "rough" categorisation, we did a second pass over the LCM searching for inconsistencies that were introduced in the process of differentiating and classifying constructs. We report on the cases we detected in three different groups.

The first group is related with *arguments* used in procedures, commands and functions. Figure 6.5 shows the concepts involved, before being modified and adapted.

- The *ArgsDeclaration* construct represents an argument declaration at the procedure level. It is only considered in Stol and Mois, even though according to our knowledge, all languages should include this construct.

LCM	Stol	Mois	Pluto	Ucl	Tope	Elisa
ArithmeticExp		X				
ArithmeticExp_Add	X		X	X	X	X
ArithmeticExp_Div	X		X	X	X	X
ArithmeticExp_Expon	X		X	X	X	X
ArithmeticExp_Mod	X				X	X
ArithmeticExp_Mult	X		X	X	X	X
ArithmeticExp_Subt	X		X	X	X	X
BoolComp		X				
BoolAnd	X		X	X	X	X
BoolNot	X		X		X	X
BoolOr	X		X	X	X	X
BoolXOr	X		X			X
Equals	X	X			X	X
GreaterEqualsThan	X	X			X	X
GreaterThan	X	X			X	X
LowerEqualsThan	X	X			X	X
LOWERTHAN	X	X			X	X
NotEquals	X	X			X	X

Figure 6.6: Some "Expressions" Inconsistencies

For the case of Pluto, the reason was that the syntax for this construct was not present in the provided grammar. In Pluto there is a different global structure called the meta-information of the mission. It is in this structure where the arguments for all the procedures are declared. We augmented the syntax of Pluto with this construct at a top level, so that it can be moved back easily to the mission meta-information when necessary. For Ucl, Tope and Elisa it was a consequence of having used more general concepts like *Arguments* and *Parameters*. We split those constructs creating the specific *ArgDeclaration* construct in those languages.

- The concepts *Parameters* and *Arguments* had the same meaning, so they were unified into *Arguments*.
- The *CommandArgs* construct was being used in Mois and Stol to recognise the arguments passed to a command. There was nothing that justifies the use of a separate construct for these kinds of arguments. It was removed by modifying the syntax of *Command* using the generic *Arguments* symbol.
- Like the previous case, *FunctionArgs* was replaced by *Arguments*, by modifying the *Function* syntax.

The second group of grammar modifications involves different kinds of boolean and arithmetic expressions. In Figure 6.6 we can see the constructs involved before being modified.

- With respect to arithmetic expressions, MOIS provides only a single generic construct, parameterised by a string. We split this monolithic construct into several specific constructs, one for each expression, therefore removing the generic *ArithmeticExp* concept.

- As in the previous case, MOIS provides a generic construct for boolean expressions. We replaced the generic *BoolComp* construct in Mois, by the specific *And*, *Or*, *Not*, and *Xor* constructs. Do notice that Pluto and Ucl do not support the *ArithmeticExp_Mod* construct. A transformation will be designed in the adaptation cycle, to adapt this mismatch with the *mod()* function.

- The *BoolNot* is included into UCL. It was simply missed from the grammar because of a human error.

- *BoolXOr* is not supported by Ucl and Tope. We will need to design a transformation to provide these two languages with a functional equivalent combination of instructions.

- Finally, the boolean comparisons like *Equals*, etc., were mistakenly omitted when building the Pluto and Ucl grammars . We included the missing constructs.

The final group of grammar modifications is a collection of different unrelated cases, that are worth to mention.

- The *Halt-Statement* construct , from Ucl and Elisa, corresponds directly to the *Exit* statement. When checking this problem, we realised also that in Ucl, the *Exit* construct was equivalent to the *Break* concept.

- The *Command* and *Expression* constructs were accidentally omitted from UCL

- In MOIS, the *Declaration* construct was equivalent to the *Declaration-Body* concept. Furthermore, a transformation was needed to collect scattered declarations into a single declarations header.

- The *Expression-List* concept, considered in Stol and Mois was not really necessary. It was redundant, and therefore removed.

- *Function-Call* was unified with *Function*.

- *Function-Name* in Stol and Mois was not necessary. It was replaced in those grammars with the more generic construct *Identifier*.

- The *PassByRef* construct used by arguments, was not being taken explicitly into account in the documentation of the Pluto, Ucl, Tope and Elisa grammars. We added it to those grammars.

- The *Wait* statement needed to be split in three different wait concepts, with a different semantics: *WaitAbsolute*, *WaitRelative* and *WaitBoolean*. This was done using program transformations.

(L2LD)	Stol	Mois	Pluto	Ucl	Tope	Elisa	LAD
Stol	0	0.06	0.17	0.10	0.08	0.09	0.10
Mois	0.06	0	0.13	0.12	0.10	0.07	0.10
Pluto	0.17	0.13	0	0.18	0.18	0.15	0.16
Ucl	0.10	0.12	0.18	0	0.04	0.10	0.11
Tope	0.08	0.10	0.18	0.04	0	0.09	0.10
Elisa	0.09	0.07	0.15	0.10	0.09	0	0.10
						LCI=	0.11

Figure 6.7: Compatibility Metrics After Correcting Inconsistencies

Finally, after the mentioned inconsistencies were detected and solved, the size of the set of language concepts was reduced by 5%, from 153 to 144 concepts. We recalculated the compatibility metrics to check how the LCM benefitted from these second pass. The new results are shown in Figure 6.7, and are analysed in Section 6.5.

Adapting the constructs: analysing mismatches and building transformations. Knowing that our priority is to translate from Stol to the other five languages, we look in the LCM for those constructs supported by Stol, that are unsupported by the rest of the languages. Then we design a draft of the transformation, and see if the constructs that will be generated are present in the other languages or not. If they are not, we sketch a different transformation, or see recursively if for the unsupported construct we can build a new transformation.

The transformations we are presenting have been simplified showing only the match and replace patterns. We are not showing other operations that can be included in the transformations, like conditional tests, to check whether or not to perform the transformation.

We present the transformations ordered from the simpler, leaving to the end the more complex adaptations.

- *Arithmetic_Mod.* Neither Pluto nor Ucl offer natively this construct. They do offer, however, a function providing the same functionality. Additionally, in both languages the function has the same name, which allows us to reuse the same transformation.

  ```
  M2F)
  Mod(lft:Expression, rgt:Expression)
  ==>
  Function(Identifier("mod"), Arguments([ lft, rgt ]))
  ```

  This is a very straightforward transformation that requires only a quick and superficial analysis of the LCM and the TCM. Both languages provide the required constructs: Function, Identifier, and Arguments.

- *BoolXOr.* This construct is absent from Ucl and Tope. They do provide, however, the boolean operations we require to emulate Xor, using And, Or, and Not.

```
X2O)
XOr(lft:Expression, rgt:Expression)
==>
Or(And(lft, Not(rgt)), And(Not(lft), rgt))
```

- *ProcedureCall* is not supported in Pluto, and there is no other native command for doing this. Nevertheless, considering that we are using the set of directives provided by an EPOCH control system, we can call one of the system directives to do the job. Indeed, many times, this is how it is done even in Stol, rather than using the native construct of the language.

```
P2D)
Proccall(id:Identifier, arg:Arguments)
==>
Dir(DirProcStart(id, arg))
```

- *While* is not supported in Elisa. Elisa fetaures only a generic loop statement that, combined with the break statement, can simulate other more specific types of loops.

```
WH2L)
While(e:Expression, b:Block)
==>
Loop(Break(Not(e)), b)
```

- Three languages do not support the *If-Else-If* construct: Mois, Pluto and Elisa. In this example, the symbol & denotes that we are requesting the execution of an operation internal to the translation system. For instance, in line 4 we request a new identifier name, that will be stored in the variable *i*. Lines 7 to 10 iterate through every item in *list*, and transform it into a series of simple If constructs.

```
IS2IF)                                                          1
Ifelsif(list:[(Expression, Block)], else:<Block>)               2
==>                                                             3
&(i = _NewId())                                                 4
LocalDeclaration(Identifier(&i)),                               5
Assignment(Identifier(&i), False()),                            6
&(foreach (e:Expression, b:Block) in list {                     7
[If(And(e, Not(Identifier(&i)))                                 8
        , Assignment(Identifier(&i), True()), b, <>),]          9
})                                                              10
If(Not(Identifier(&i)), else, <>)                               11
```

- *Wait*. This case was already explained in detail in Section 5.2.4.1, when we showed how to use the TCM table to link different related transformations to reuse them. Three transformations were designed and put together, solving the mismatches for the six languages, and the set of three language concepts related with the *Wait* functionality.

- *Goto - Label* This is an example of a mismatch for which our proposed automatic solution did not work for all the procedures, because the algorithm we used is exponential. For some procedures we did not have enough resources to complete the process. For other procedures, the resulting code was too big.

  In our language family, the Goto and Label constructs are provided natively by Stol and Elisa. In the case of Mois gotos are deprecated, but it is still possible to use them. The gotos are not supported in Pluto, Ucl and Tope.

  The case of gotos in Stol is particularly interesting. Stol does not enforce any restriction on the kind of jumps that can be introduced in the code. It is possible to jump inside and outside of any block of code (loops included), and across different nesting levels. Jumping inside a loop is specially problematic, because it provokes irreducible regions in the control-flow [80]. Irreducible regions are complex to solve for any goto removal strategy. Additionally, for some of the Stol test procedures, the irreducible region covered more than 80% of the code in the procedure. Finally, Stol does not support the constructs required by some of the existent goto removal strategies.

  Many strategies have been proposed to solve the goto removal problem. In [23], Ceccato et al. report on five of them. Ceccato compares these strategies in the context of a migration project from a legacy language to Java:

  – In the pattern-based strategy [122], recurring goto patterns are identified by the programmer. The programmer, then, designs an equivalent replacement pattern without gotos. These patterns are implemented as rewriting rules that automatically translate the code recognised by the pattern. For some tangled goto structures, like was the case with some of the Stol test procedures, we could not define the required patterns. Ceccato reports a successful removal of only 21% of the gotos in the code he analysed.

  – The Bohm-Jacopini strategy [16], and the Erosa strategies [41], cannot be used in our experiment. Both of them need constructs that are not supported by Stol: Bohm's strategy uses Switch and Continue constructs, and Erosa's strategies use Break and Continue constructs.

  – Finally, the JGoto strategy was specific to Java, and worked only at the java byte-code level.

  A different strategy, not considered in Ceccato's experiment is the node-splitting strategy [80], which normalises the irreducible regions in a control-flow graph by iteratively applying the following three functions:

  – T1) Remove edges connecting a node to itself.

  - T2) If a node has a single predecessor, merge it with its predecessor, preserving their incoming and outgoing edges.
  - T3) If a node has multiple predecessors, duplicate it to produce one copy per predecessor.

  The process continues until there is one single node. If this process is reversed, preserving the duplicated nodes, the result is a reducible control-flow graph.

  This strategy does not introduce additional constructs, or different kinds of jumps in the program, and uses a single algorithm for every program, which makes it our best candidate for removing the gotos from Stol procedures. The disadvantages, though, are important: the algorithm is exponential. When the irreducible regions are too big, two scenarios become possible:

  - The resulting code can be considerably larger than the original.
  - The algorithm can consume all the available resources in the computer, and stop before completion.

  We finally decided to implement and try the node-splitting strategy. It is general enough to be used with languages like Stol, and it has the potential to be improved with, for instance, a heuristics approach, as proposed by Unger et al. [116]. We explored Unger's approach, producing a semi-automatic version of node-splitting: for some procedures it is possible to duplicate and split a few selected nodes, from the post-dominator of the control-flow graph, before applying the node-splitting algorithm. This reduces the size of the irreducible region and produces smaller programs.

- The *Telemetry* construct deserves special attention. According to the LCM table, Stol and Ucl do not provide this construct. Nevertheless, telemetries, together with commands and directives, are instructions that any procedure willing to communicate with the control centre or the spacecraft needs to use.

  The explanation for this situation is that Stol and Ucl hide the activity of fetching a telemetry value. In both cases the interpreter keeps track of all the identifiers declared with a *Point Declaration*. Then, every time they are used in the code, the compiler fetches the value of the telemetry behind scenes. The rest of the languages in the family use a more "visual" approach, thanks to the specific telemetry constructs.

  To solve this case we needed to introduce artificially the *Telemetry* construct in Stol, when the target languages natively implement the specific *Telemetry* construct. The transformation basically collects all the statements using a point identifier, and inserts before them the corresponding *Telemetry* construct.

```
i_TLW)
&(list:[Statement, [Identifier]]
      = CollectStatementsHavingPointIdentifiers())
==>
&(foreach([s:Statement, listid:[Identifier]] in list) {
 foreach(id in listid) {
  Telemetry(id), s
 }
})
```

This *inverse* application of the transformation is peculiar, in the sense that it applies when the source language does not provide a construct. In general we need to adapt when the target language does provide the construct.

- Another case of *inverse* application of a transformation, involves the *Assignment-to-Tlm* construct. It is possible in certain cases to update a telemetry temporarily with an arbitrary value.

  Stol makes no distinction between this case and a regular assignment. Mois and Pluto, however, use a special instruction for those cases where an arbitrary value must be assigned to a telemetry.

```
i_TLAS)
Assignment(t:Telemetry, e:Expression)
==>
AssignToTlm(t, e)
```

- The last mismatch we need to solve relates to how declarations can be scattered or not in the procedure. Stol allows to declare variables anywhere in the code. Mois, Pluto and Elisa require that variables are declared into the *DeclarationsBody* header, at the beginning of the procedures.

```
D2DB)
&(list:[Declaration] = CollectAndRemoveAllDeclarations())
==>
&(addfirst(Start, DeclarationsBody(list)))
```

In Figure 6.8 we can see the LCM table after having included all the transformations we have designed. We repeated only the language concepts that were affected.

Finally, we recalculate the compatibility metrics to check how the LCM benefited from these transformations. The results are shown in Figure 6.9.

Generating the translators. Before generating the translators, it is necessary to decide the order in which the additional transformations we have designed must be applied. The constraint that has to be respected is that a transformation modifying some construct C, must always be applied after any other transformation that generates the same construct C.

LCM	Stol	Mois	Pluto	Ucl	Tope	Elisa
ArithmeticExp_Mod	1	1	M2F	M2F	1	1
Assign-To-Tlm	i_TLAS	1	1	i_TLAS	i_TLAS	i_TLAS
BoolXOr	1	1	1	X2O	X2O	1
Declaration	1	D2DB	D2DB	1	1	D2DB
If-Else-If-Statement	1	IS2IF	IS2IF	1	1	IS2IF
ProcedureCall	1	1	P2D	1	1	1
Telemmetry	i_TLW	1	1	i_TLW	1	1
Wait-Boolean-Statement		1	1	WB2RU		1
Wait-Relative	1	WR2WB	1	WR2WB	1	1
Wait-Absolute	1	WA2WR	1	WA2WR	WA2WR	1
While-Statement	1	1	1	1	1	WH2L

Figure 6.8: LCM with transformations

(L2LD)	Stol	Mois	Pluto	Ucl	Tope	Elisa	LAD
Stol	0	0.02	0.13	0.08	0.06	0.06	0.07
Mois	0.02	0	0.10	0.07	0.07	0.04	0.06
Pluto	0.13	0.10	0	0.12	0.13	0.12	0.12
Ucl	0.08	0.07	0.12	0	0.03	0.04	0.07
Tope	0.06	0.07	0.13	0.03	0	0.06	0.07
Elisa	0.06	0.04	0.12	0.04	0.06	0	0.06
						LCI=	0.08

Figure 6.9: Compatibility Metrics After Adding Transformations

We use the information inside the TCM to calculate this order, as explained in Section 5.2.4.1. Figure 6.10 shows the TCM for this example. The transformations are already ordered from left to right. The system will start with the left-most transformations. The cells shadowed in gray show the sections of the table that specifically affect the order. A value of I indicates that the transformation modifies that construct, and a value of X that the translation generates that construct.

Finally, not all the transformations have to be included in every translator. Figure 6.11 shows the list of transformations considered in every translator having Stol as source language.

Translating the test procedures. The translation was performed in three different steps. First, we did a test translation from Stol to Tope, with the original procedures. We confirmed that all the Goto and Label constructs where correctly detected and tagged by the generated translator as mismatches, because Tope does not support the Goto construct. In total 117 cases where signaled in the translated code, with the "Mismatch" tag:

```
...
<Mismatch from="Stol" concept="Goto">
  goto EXIT
</Mismatch>
...
<Mismatch from="Stol" concept="Label">
  EXIT:
```

	X2O	P2D	WH2L	M2F	IS2IF	WA2WR	WR2WB	WB2RU	I_TLW	I_TLAS	D2DB
Arguments				X							
ArithmeticExp_Mod				I							
ArithmeticExp_Subt						X					
ArithmeticExp_Add							X				
Assign-To-Tlm										X	
Assignment					X		X			I	
BoolAnd	X										
BoolNot	X		X								
BoolOr	X										
BoolXOr	I										
Break-Statement			X								
Declaration					X		X				I
Declarations-Body											X
Directive		X									
Directive_ProcStart		X									
Equals					X						
Function				X		X	X				
GreaterThan							X				
Identifier				X	X	X	X		I		
If-Else-If-Statement					I						
If-Statement			X		X						
Loop-Statement			X								
ProcedureCall		I									
Repeat-Statement								X			
Start											X
Telemmetry									X	I	
Wait-Absolute						I					
Wait-Boolean-Statement							X	I			
Wait-Relative						X	I				
While-Statement			I								

Figure 6.10: TCM Recommendations

(From Stol to ...)	X2O	P2D	WH2L	M2F	IS2IF	WA2WR	WR2WB	I_TLW	I_TLAS	D2DB
Mois					X	X	X	X	X	X
Pluto		X		X	X			X	X	X
Ucl	X			X		X	X			
Tope	X					X		X		
Elisa			X		X			X		X

Figure 6.11: Transformations per translator

				Goto Removal							
	ORIGINAL			Manual		Semi-Automatic			Automatic		
	# Goto	# Label	# Lines	Lines	Diff.	Lines	Diff.	Time(sec)	Lines	Diff.	Time(sec)
P01	5	1	614	644	5%	1069	74%	7.6	4907	699%	21.2
P02	25	8	827	1194	44%						
P03	6	6	274	286	4%	273	0%	2.2	383	40%	2.4
P04	0	0	198								
P05	6	4	308	344	12%	505	64%	4.4	2027	558%	9.9
P06	6	3	996	1033	4%	2453	146%	12.5	10278	932%	243.7
P07	4	4	451	484	7%	792	76%	6.6	1369	204%	25.6
P08	23	14	972	1248	28%	4188	331%	68.9			
P09	0	0	152								
P10	1	1	102	108	6%	89	-13%	9.9	89	-13%	9.1
	76	41	4894	5341	14%	9369	97%	112.1	19053	403%	311.9

(This experiment was executed on a Mac Pro 8-Core, with 15GB of RAM).

Figure 6.12: Goto removal. Methods comparison.

```
</Mismatch>
...
```

The mismatch tag is used when any symbol that cannot be understood by the target language grammar is found. If it is possible, after tagging the mismatch, the translator tries to translate the rest of the code.

Second, to remove gotos and labels we tested three different alternatives, whose results are summarised in Figure 6.12:

- a) We translated all the procedures by hand, using mainly code duplication to solve the improper loops. We did it this way to obtain a better understanding of the structure of the programs, and as a backup solution: the node-splitting algorithm we use for the automatic goto removal is exponential and for some programs the solution will not be reached. The best result in terms of the size of the code produced, compared with the other two techniques, were obtained with this manual approach, with an increase of 14% in the number of lines of code.

- b) We tried the semi-automatic translation, with the assisted version of node-splitting, explained previously in this section. With this technique, the procedure P08 could not be processed. The duplication of nodes required by the node-splitting algorithm exhausted all the memory in the computer. With the semi-automatic method, the number of lines of code increased 97% in average. The total time used to process the 7 procedures that succeed to complete, was 112 seconds.

- c) We used the automatic translation with the node-splitting algorithm. Two procedures, P02 and P08, could not be processed with this method. The number of lines of code increased by 403% in average. The six procedures were processed in 311 seconds in total.

In all three cases, we used the bisimulation tool to confirm the weak bisimilarity between the original procedure, and the procedure without the goto-label constructs.

In the third step, to proceed with the final translation, we decided to use the manually generated "goto free" procedures. The decision was based in their smaller code, compared to the size of the code generated by the semi-automatic and the automatic strategies.

The translation completed successfully for the ten procedures and the five target languages. One final step was performed as an additional test for completeness: we translated the generated procedures back to Stol again. The following actions were necessary for this final step:

- First, we created four new transformations to revert the changes suffered by the procedures.
  - TLAS, which reverts the effect of the i_TLAS transformation. This transformation was necessary to translate from Mois and Pluto. It transforms the AssignToTlm construct into the Assignment construct.

    ```
    TLAS)
    AssignToTlm(t:Telemetry, e:Expression)
    ==>
    Assignment(t, e)
    ```

  - DB2D, which reverts the effect of D2DB. This transformation applies when translating from Mois, Pluto and Elisa. The transformation replaces the DeclarationsBody construct by each Declaration construct contained in the list inside DeclarationsBody.

    ```
    DB2D)
    DeclarationsBody(list:[Declaration])
    ==>
    &(foreach(d in list) {
          d
    })
    ```

  - TLW, which reverts the effect of i_TLW. This transformation applies when translating from Mois, Pluto, Elisa and Tope. This transformation simply removes the Telemetry construct. Remember that in Stol, fetching the value of a telemetry is done automatically by the compiler, when a variable referencing a telemetry is used.

    ```
    TLW)
    Telemetry(id:Identifier)
    ==>
    //
    ```

  - L2GOTO, which reverts the effect of WH2L. This transformation applies when translating from Elisa. This transformation does not revert the Loop construct to a While, so it cannot be considered strictly as inverting the effect of WH2L. Nevertheless it produces functionally equivalent code, and it transforms any Loop construct to Stol. Unfortunately it cannot be used to translate from Elisa to

Pluto, Ucl or Tope, because these three languages do not support the Goto construct.

```
L2GOTO)
Loop(a:[Statement], Break(e:Expression), b:[Statement])
==>
&(in = _NewId(); out = _NewId())
Label(&in),
a,
If(e, Goto(&out), <>),
b,
Goto(&in),
Label(&out)
```

Only four transformations needed to be reverted. For the other 6 transformations used when translating from Stol (M2F, X2O, P2D, IS2IF, WR2WB and WA2WR), they already produced constructs supported by Stol. The translators built this way are not complete, because they do not consider all the incompatible constructs that need to be transformed. The translators were, nevertheless, complete enough as to translate back to Stol the tested procedures.

Verifying the equivalence. The verification tool proved to be useful not only as the final step in the translation, testing the resulting procedures. It is useful also while building the translators, as a debugging tool. In our experiment, more specifically, for every transformation we used a set of small test programs that were tested for equivalence after every important modification of the transformations. It helped to detect and correct errors introduced by accident in the logic of the transformations.

Next, we present some more specific cases detected when testing the translators.

- Verifying the weak functional equivalence of two procedures from different source code files can be done with some restrictions in the kind of tags used by the control-flow graphs. It is necessary to use non-strict tagging, because the unique identifiers of the nodes in the two ASTs are no longer aligned. This was the case when we use the verification tool to check the weak equivalence between the original Stol procedures, and the procedures where the gotos were removed by hand.

- Similar to the previous problem, whenever you need to create a new instruction that will be tracked during the verification, or when you delete one such instruction, the verification will not be able to establish a relation and will stop.

  One such example happens when translating from Stol to Mois. Stol uses the *WaitAbsolute* construct, which need to be transformed into a *WaitRelative* construct in Mois. The two constructs, become unrelated

for the verification tool. This case, though, is simply solved by renaming the two instructions into a common WAIT tag, when producing the CFG, even though precision will be lost.

More difficult to solve becomes the case when we replace the original instruction with something we no longer track. We did not suffer this problem in this experiment, but if we want to translate from UCL to Tope, the WaitBoolean construct will be changed with a loop, and loop constructs disappear as individual entities in the CFG.

An alternative could be to use some special notation for these cases, when tracked constructs changed radically. This way the verification tool would identify those cases as signaled exceptions, and continue.

- The verification is sensible to the branching structure in our programs, and false negatives can appear when we modify this structure disregarding this sensibility. Let us illustrate this case with a transformation required when translating from Stol to Elisa. Stol supports the *If-Elseif* construct like in the following example:

```
X
IF (a) THEN
   Y1
ELSEIF (b)
   Y2
ELSEIF (c)
   Y3
ELSE
   Y4
ENDIF
Z
```

Elisa does not support this construct, so we build a transformation replacing the *If-Elseif* construct, with a non-nested structure of *If* constructs, using one auxiliary boolean variable to control the branching, as shown in the following piece of code:

```
my_0 = FALSE
X
IF ( a && ! my_0 ) THEN
 my_0 = TRUE
 Y1
ENDIF
IF ( b && ! my_0 ) THEN
 my_0 = TRUE
 Y2
ENDIF
IF ( c && ! my_0 ) THEN
 my_0 = TRUE
 Y3
ENDIF
```

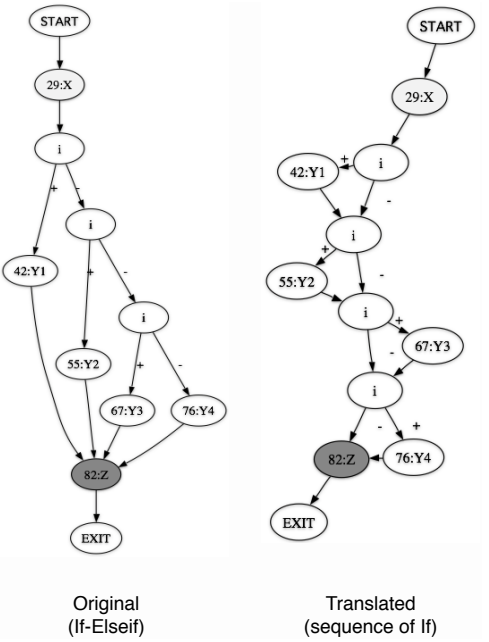


Figure 6.13: Bisimulation False Negative

```
IF ( ! my_0 ) THEN
 Y4
ENDIF
Z
```

Even though these two pieces of code execute the same, the verification shows an inconsistent path: in the translated program it is possible to go directly from the execution of *X* to the execution of *Z*, as shown in Figure 6.13.

The verification stops when it finds the first inconsistency. In this case, though, it is easy to see that many inconsistent paths exist, like for instance from *Y1* to *Y3*.

To avoid this inconsistency, we changed the implementation of the transformation into a nested *If* structure, that more naturally resembles the original structure. Also for the end-user of the program, this alternative will be easier to recognise and to identify with the old one.

```
X
IF ( a ) THEN
  Y1
ELSE
  IF ( b ) THEN
    Y2
```

```
  ELSE
    IF ( c ) THEN
      Y3
    ELSE
      Y4
    ENDIF
  ENDIF
ENDIF
Z
```

For cases where we cannot provide with an alternative implementation for the transformation, we cannot use the verification on procedures presenting these constructs.

Impact on reuse. Our global experiment consisted of creating a language family of six languages: Stol, Mois, Pluto, Ucl, Tope and Elisa. This was achieved by adding a total of 770 lines of specifications (annotations) to the grammars of the six languages. Then, we built 10 translators in total: 5 from Stol to the other languages, and 5 to bring the translated code back to Stol. Thanks to the specifications included in the grammars, 76% of all the productions in the family (109 / 144) could be mapped directly between the six languages in the family. The Stol language consists of 120 productions. For the 11 Stol productions that could not be mapped directly with the other five languages, a total of 15 additional functions where added to the language family structure for solving the mismatches. Each function was used in 2.27 translators in average (ranging from 1 to 4 translators).

Compared with a more traditional approach, where translators are built independently without an explicit categorisation and mapping of compatible constructs, the effort would have been considerably different.

First let us consider the case of those productions that were directly mapped together between the six languages. For translating one of these compatible productions, in our approach we need to write one line of specifications per language in average: 6 lines of code for the 10 translators we built. With a different approach, like the one we mentioned in the previous paragraph, this would have required 10 functions in total, one per translator. These functions are in general not complicated, and they require 3 lines of code in average. Even considering that defining a correct mapping can be a more difficult task than programming any of these 10 simple functions, still the effort measured in lines of code is five times smaller: 6 vs. 30 lines of code per mapped production.

Second, for the functions that need to be programmed to adapt the mismatches, it is more difficult to give a generic view of the improvement.

It largely depends on the complexity of every case. For this experiment though, instead of only 15 functions, we would have needed to write 34, one for each mismatch in every translator, which means that the total effort, measured in lines of code, was reduced by more than the half.

We believe that these facts show that an important degree of reuse can be achieved, thanks to the product-line structure implemented, from every specification and function that we add to the language family. Even though the effort of producing the specification and the functions is not negligible, it pays in the short term, when many translators are expected to be built.

6.5 Discussion

Construct differentiation requires to abstract away, from the concrete grammar used for parsing, specific details linked more with the implementation of the parser, than with the semantics of the construct. It is a step that requires a balance between a concrete and an abstract grammar. We cannot simply get rid of all terminal in the productions, because enough syntactic information needs to be left, that can explain the behaviour of the construct. This syntactic information we leave in the constructs is required for the use of the classification assistant.

Classifying the constructs into the LCM is an iterative process. When we put the constructs together into the family, many inconsistencies can be shown, like involuntary omissions of constructs in the grammar and duplicated concept definitions. For cases where more languages are considered, or if the languages have considerably more productions, we can assume that more than two revisions of the table are required.

The classification assistant provided a good percentage of usable recommendations: for 85% of the productions the correct recommendation was found in the first two positions. The results may somehow be biased though, because the grammars were analysed and manipulated by the same person, during a reasonably short period of time. This probably stimulate that the symbols were named following the same schema. On the one hand we probably cannot expect the same results if the grammars are preprocessed by different persons. On the other hand, this possible bias reinforces the idea that if we follow an organised set of criteria, the grammar alignment performed during the differentiation process is effective.

With respect to the adaptation step, the goto-label issue remains unsolved. Not being able to provide an automatic solution that works with every program, hampers the process of translation. Nevertheless, this is not a problem of the approach to build translators, but rather of our ability to come up with a more effective algorithm. A positive point is that this problem allowed us to show that we can also use a refactoring process, previous to the transla-

tion, and still get results that can be verified afterwards, even though with a reduced precision.

In this experiment we were able to define one single order to apply the transformations, that works with all the procedures. No cycle between transformations was found, partly because there is a small number of transformations. For other cases, if cycles are detected, the user will receive a warning about the transformations involved, and the conflicts will need to be solved by hand. Generating the order for applying the transformations for each translator independently, reduces the number of transformations to consider, and reduces the probability that cycles will show up.

The verification step is very useful when testing the designed translations. Problems that slip from the designer's mind can be caught by verifying the transformations on simple examples. While defining and testing the transformations, we used the verification after every iteration, and many problems in the logic of the transformations were detected.

The verification is difficult to use when the sensitive operations we track in the CFG, need to be modified by the transformations, producing a different instruction. As annoying as this is, it is nevertheless an expected behaviour: the very principle behind the verification is that the sensitive instructions must not change. Which is why we check them during the bisimulation. To overcome this problem, there is always the resource of renaming the instructions. If renaming does not apply, we should reevaluate if the instructions involved need to be modified, or tracked.

More restrictive is the problem between verification and modifications to the branching structure of the program. As we could see, false negatives appear when non existing problems are detected. The real problem though, is that we cannot check what happens inside the conditions that control branching. If, for instance, we omit by mistake a boolean modifier, the verification will never detect it, and will give a false positive, much more dangerous than a false negative.

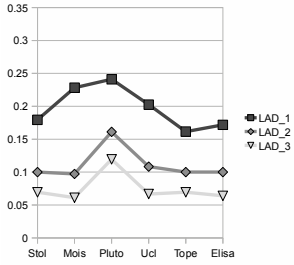


Figure 6.14: LAD Evolution.

Finally, the use of metrics shows how the actions put in practice during every step have a positive effect reducing the entropy of the system, as shown in Figure 6.14, which summarises the information presented in figures 6.4,

6.7 and 6.9, where we presented the measures taken after classifying the constructs, after a second iteration to check for inconsistencies, and after defining the transformations.

The difference between the first measure, LAD_1, taken after the languages were classified into the LCM, with respect to the second measure, LAD_2, taken after a second revision for classification inconsistencies, shows that simple efforts aligning language concepts have a significant impact in the languages compatibility. The effect is less dramatic comparing LAD_2 with LAD_3, taken after the adaptation with program transformations, because not all the incompatibilities were solved. An improvement is shown, nevertheless.

Regarding the choice of Stol as the first language included in the family, we can see in LAD_1 that even if it does not have the best Language Average Distance, it is among the better ranked (the lower the value of the entropy, the better in terms of compatibility). LAD_2 confirms the choice. Other languages could have been good candidates as well, like Tope. The case of Pluto is interesting because it persists as the less compatible language in the three measures. This lower compatibility, is also a consequence of having more constructs than the other languages. Anyway, the compatibility of Pluto is also improved at each new step.

6.6 Conclusion

Thanks to this experiment we have been able to go through all the necessary steps to build a product-line for a family of programming languages. We generated a group of program translators from Stol to Mois, Pluto, Ucl, Tope and Elisa, and we evaluated the translators using a group of test Stol procedures. The experiment covers all the steps proposed in our methodology.

The approach shows some limitations, that are manifested for instance with the goto-label problem, due to its exponential algorithm, and with the false negatives found during verification, because our weak bisimulation approach is oblivious to the values of expressions.

Advantages of the approach are also shown. Among others that the use of the LCM and TCM common structures, provide an effective way to organise language concepts and reuse program transformations. We also confirmed that the verification based on weak bisimulation is a useful technique to detect problems with the transformations and with the translation in general.

The methodology, the techniques, and the tools that are grouped together, allowed us to build the required family of program translators.

7 Conclusion

7.1 Summary

Throughout this thesis we presented our methodology for building families of program translators. This methodology is based on the following pillars:

- Product-lines. This software engineering approach provides the basic architectural principles that allow us to put together a variety of tools and techniques, making them work collaboratively.
- Grammarware. Our work required not just an operational software engineering methodology, but probably more important, a solid language engineering orientation guiding us along the full process, and structuring our efforts.
- Families of languages. By grouping the languages to translate into a family, where all its members share a common set of constructs and common semantic foundations, we reduce to a realistic dimension an otherwise very large problem.
- A set of existing, third-party, state-of-the-art tools for the analysis, manipulation and verification of programs and programming languages.

Thanks to these pillars we showed how to:

- Build a generic framework to produce families of program translators. The product-line approach, and its emphasis on clearly identifying the commonalities and variabilities between the different elements in our family, helped us in defining simple structures like the Languages Concepts Matrix and the Transformations Concepts Matrix. Organising the languages into these structures was fundamental to the generation of the final family of translators.
- Align the languages and make them converge into language family. Grammarware principles and methodologies were applied to go from language documentation, to grammars to a language family. The original language documentation is processed such that successively applying grammar recovery and convergence techniques, we can produce the required grammars for parsing, classifying and verifying.
- Reuse program transformations. Having a common "dictionary" of concepts, and designing the transformations based on these common

concepts, makes it straightforward to combine existing transformations into new translators.

- Verify a weak form of functional equivalence between original and translated programs. Thanks to the use of a lightweight semantic annotation technique, we can automatically obtain the input for a simplified verification tool, that performs an initial assessment of the correctness of the translation.

- Produce the required family of translators. By grouping the languages into a family sharing a common set of constructs and transformations, we can provide an automated generic approach which can be used to build translators between any pair of those languages.

7.2 Contributions

The main contribution of this work is its integrated approach to build families of program translators. Different techniques and methodologies, coming form different domains, are combined into a consistent framework. Specific contributions we would like to mention are the following:

- A product-line oriented architecture dedicated to families of programming languages, for the production of program translators.

- A strong use of regular annotations in SDF grammar definitions, to automatically produce preliminary descriptions for language tools.

- A technique for language concepts categorisation based on:
  - A set of guidelines to preprocess grammars for alignment.
  - An assisted classification technique, based on the syntactic pattern of grammar productions.

- An explicit technique for transformation rules sequencing, based on partial ordering, and topological sort.

- A domain-specific language to define lightweight control-flow semantics on language grammars, and to automatically generate the control-flow graphs of programs.

- An application of weak bisimulation for the lightweight verification of functional equivalence between original and translated programs.

- A specific set of metrics, based on the concept of entropy reduction, that facilitates the process of evaluating how the different actions we take to adapt the languages' compatibility, affect the organisation of the language family system.

7.3 Limitations

Despite the contributions we mentioned, some limitations of the approach have been identified. We present them now.

- Languages in the family have to be semantically very similar to obtain some benefit of the additional work of indexing the grammars into the common family structure. If the languages are too different, first, the grammar alignment is limited and more program transformations will need to be implemented. Second, which is a consequence of the first point, the programmed transformations rely more on unaligned concepts, and the possibilities of reusing these transformations are therefore reduced. The advantage of the "economy of scope" provided by the product-line approach disappear.

- The verification approach we use can only analyse the ability of one program to follow the same paths of execution of another program. The verification is blind to the result of expressions.

  This limitation is specially restrictive for the case of boolean expressions controlling the branching in control flows. We cannot know if we are entitled to follow a path, because we do not know all the associated conditions that enable to do it.

  This blindness to expressions may provoke false positives when the paths have been inverted, or false negatives when a path exists even if it cannot be followed because of some boolean restriction.

- The approach was validated with only one language family. We cannot be sure that the methodology can be extended to other families while preserving the same characteristics and advantages. The scope of the technique has not been clearly established yet, and its generality is assumed but not confirmed.

- The solution we implemented to solve the Goto removal problem is generic, but is not effective. The node-splitting algorithm that is used is exponential. For programs were the irreducible region is too big or contains nested irreducible regions, it would fail if we do not have enough ressources to calculate the solution. Moreover, the size of the code of the programs that can be transformed, tend to be considerably bigger than in the original program. This increase in code size makes harder to manipulate the resulting program.

7.4 Future Research

Our work focused on providing a methodology for building families of program translators. In the process of doing so, many questions and additional

problems were unveiled. Some we were able to solve, others, that we mention next, still need an answer.

- The use of weak bisimulation to verify the functional equivalence of programs is promising. Several alternatives could be envisaged to develop a more robust and parameterisable technique:
  - Augment the number and type of instructions to be checked. This implies augmenting the number of transitions in the LTS, but that is something the CADP Bisimulator seems to handle without any problem, eliminating one of the technical problems that similar studies were having [31]. The problem is, however, that the more nodes we add to the graph, the more we limit the liberty that transformations will have to manipulate the code. For instance, let us suppose we decide to include in the control-flow graph one node for every *Assignment* in a program. This immediatly implies that the bisimulation will fail for any transformation introducing auxiliary variables to control the flow, like we do in Section 6.4 with the IS2IF transformation that translates the If-Else-If construct to a sequence of If constructs: the transformation will create a series of assignments that do not exist in the original program. A good compromise between structure and functionality is necessary.
  - How to combine bisimulation with other techniques, in an organised workflow, that alternate each other when necessary. Coming back to our false negative example from Section 6.4, if we combine bisimulation with some form of data-flow analysis, we could see that the path reported as inconsistent by the bisimulation cannot be reached in practice thanks to the boolean guard introduced by the transformation.
  - How to extend the technique we use to generate the CFGs with other primitives that can translate functionality into states, for instance when manipulating boolean conditions.
- Language concepts classification is merely based on syntactic features. Other features though do not appear as constructs at all. In the validation experiment presented in Section 6.4, we have been able to translate into constructs, the functionality provided by the languages that was relevant to our problem. For other families, it could not be the case. It is necessary to study how the language family structures like the LCM and the TCM can be augmented to handle non-context free restrictions that currently need to be dealt with transformations.
- Other families of programming languages could benefit from this approach, but more experimentation is needed. Different applicative domains need to be analysed, as well as programming languages with different paradigms.

- The metrics we have proposed, and the use we did of them, are still rather limited. Especially if we are interested in extending the study to other families of languages, it is necessary to measure how common or different these families and languages are, and to assess the validity of the metrics proposed.

- Grammar annotations require a full study on their own. We have successfully used simple annotations to align grammars and to produce automatically large parts of the translators. Our work with the CFSL showed how to use more advanced annotations to generate CFGs that can be analysed for some weak kind of functional equivalence. This, we believe, are only two small examples of how a grammar can be extended beyond syntax to provide with a better description of languages.

- We have used annotated grammars to semi-automatically produce translators. Other kinds of tools for program visualization, analysis or manipulation, could be described and generated using this approach as well.

- In our approach, the verification between original and translated procedures is done after the translation. We could benefit of some form of verification, before the translation, on the implemented transformations. It could be similar to what Veerman did in his work with Cobol program transformations [122]. Even more, a lightweight form of weak bisimulation could be applied on the grammar graphs of the languages we want to translate. We can learn more about possible incompatibilities that are deep in the nested structure of the languages, even before implementing the transformations

Bibliography

The references are sorted alphabetically by first author.

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference, 1972. ISBN 0139145567.

[2] A. V. Aho and J. D. Ullman. Translations on a context-free grammar. In *STOC '69: Proceedings of the first annual ACM symposium on Theory of computing*, pages 93–112, New York, NY, USA, 1969. ACM Press.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[4] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 114, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8377-5.

[5] M. H. Alalfi, J. R. Cordy, and T. R. Dean. SQL2XMI: Reverse engineering of UML-ER diagrams from relational database schemas. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 187–191, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3429-9.

[6] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag.

[7] ASTRIUM. User control language reference manual. http://www.astrium.eads.net/, 2003.

[8] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

[9] G. J. Badros. JavaML: a markup language for java source code. In *The International Journal of Computer and Telecommunications Networking*, pages 159–177. North-Holland Publishing Co., 2000.

[10] E. Balland, P. Brauner, R. Kopetz, P. E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Term Rewriting and Applications*, Lecture Notes in Computer Science, pages 36–47. Springer-Verlag, 2007.

[11] H. Basten and P. Klint. Defacto: Language-parametric fact extraction from source code. pages 265–284, 2009.

[12] I. D. Baxter. DMS: Program transformations for practical scalable software evolution. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 48–51, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-545-9.

[13] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. Bisimulator: A modular tool for on-the-fly equivalence checking. In N. Halbwachs and L. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland)*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer Verlag, April 2005.

[14] M. Bernardo and S. Botta. A survey of modal logics characterising behavioural equivalences for non-deterministic and stochastic systems. *Mathematical Structures in Computer Science*, 18(1):29–55, 2008. ISSN 0960-1295.

[15] A. Blass, N. Dershowitz, and Y. Gurevich. When are two algorithms the same. *The Bulletin of Symbolic Logic*, 15(2):145–168, 2009.

[16] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. pages 11–25, 1979.

[17] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-67494-7.

[18] S. Botta-Dukat. Rao's quadratic entropy as a measure of functional diversity based on multiple traits. *Journal of Vegetation Science*, 16 (5):533–540, 01 2005.

[19] J. Brant and D. Roberts. SmaCC, a smalltalk compiler-compiler. http://www.refactory.com/Software/SmaCC.

[20] S. Buss, A. Kechris, A. Pillay, and R. Shore. The prospects for mathematical logic in the twenty-first century. *The Bulletin of Symbolic Logic*, 7:169–196, 2001.

[21] J. Calera-Rubio and R. C. Carrasco. Computing the relative entropy between regular tree languages. volume 68, pages 283–289, Amsterdam, The Netherlands, 1998. Elsevier North-Holland, Inc.

[22] Y. Cao. A hierarchy of behavioral equivalences in the pi-calculus with noisy channels. *The Computer Journal*, 53(1):3–20, 2010.

[23] M. Ceccato, P. Tonella, and C. Matteotti. Goto elimination strategies in the migration of legacy code to Java. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2157-2.

[24] N. Chomsky and M. Schutzenberger. The algebraic theory of context-free languages. *The Journal of Symbolic Logic*, 32(3):388–389, 1967.

[25] A. Christoph. Graph rewrite systems for software design transformations; objects, components, architectures, services, and applications for a networked world. In *International Conference NetObjectDays, NODe 2002*, pages 7–10, 2002.

[26] D. P. Clark, M. Chen, and J. V. Tucker. Automatic program translation - a third way. *International Symposium on Multimedia Software Engineering*, 0:265–272, 2004.

[27] T. Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussel, 2007.

[28] A. Cleve and J.L. Hainaut. Co-transformations in database applications evolution. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 409–421. Springer, 2006. ISBN 3-540-45778-X.

[29] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[30] J. R. Cordy. TXL - A Language for Programming Language Tools and Applications. *ENTCS*, 110:3–31, 2004.

[31] M. J. S. Pelican D. J. Musliner and P. J. Schlette. Verifying equivalence of procedures in different languages: preliminary results. *VV&PS 2009*, 2009.

[32] J. Daintith. A Dictionary of Computing. http://www.encyclopedia.com/, 2004.

[33] G. de Geest, S. D. Vermolen, A. van Deursen, and E. Visser. Generating version convertors for domain-specific languages. In Andy Zaidman, Massimiliano Di Penta, and Ahmed Hassan, editors, *Proceedings 15th Working Conference on Reverse Engineering (WCRE 2008)*, pages 197–201. IEEE Computer Society, 2008.

[34] J. DeBaud and K. Schmid. A systematic approach to derive the scope of software product lines. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 34–43, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0.

[35] B. Demeuse and S. Valera. Pluto, a procedure language for users is test and operations. *DAta Systems In Aerospace, DASIA*, pages 307 – 310, 1998.

[36] V. Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995. ISBN 9810220588.

[37] D. Dougherty and A. Robbins. *SED & AWK (2nd Edition)*. O'Reilly Media, Inc., 1997. ISBN 1565922255.

[38] E. B. Duffy and B. A. Malloy. An automated approach to grammar recovery for a dialect of the C++ language. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 11–20, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3034-6.

[39] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

[40] H. R. Elliotte. *XML Bible*. Hungry Minds, gold edition, 2001.

[41] A. M. Erosa and L. J. Hendren. Taming Control Flow: A Structured Approach to Eliminating GOTO Statements. In *International Conference on Computer Languages*, 1994.

[42] European Cooperation for Space Standardisation. Test and operations procedure language. http://www.ecss.nl/, 2006. ECSS-E-70-32A.

[43] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. pages 296–309, 2000.

[44] K. N. Ganeshaiah and S. Uma. Measuring biological heterogeneity of forest vegetation types: Avalanche index as an estimate of biological diversity. *Biodiversity and Conservation*, 9(7):953–963, 07 2000.

[45] K. N. Ganeshaiah, K. Chandrashekara, and A. Kumar. Avalanche index: A new measure of biodiversity based on biological heterogeneity of the communities. *Current Science*, 73(2):128–133, 07 1997.

[46] K. N. Ganeshaiah, K. Sagar, and S. Uma. Floral resources of Karnataka: A geographic perspective. *Current Science*, 83(7):810–813, 10 2002.

[47] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. *Computer Aided Verification*, pages 158–163, 2007.

[48] J. Gray, J. Zhang, Y. Lin, S. Roychoudhury, H. Wu, R. Sudarsan, A. Gokhale, E. Neema, F. Shi, and T. Bapty. Model-driven program transformation of a large avionics framework. In *Avionics Product Line Architecture, Generative Programming and Component Engineering (GPCE 2004), Springer-Verlag LNCS*, pages 361–378. Springer-Verlag, 2004.

[49] M. L. Griss. Software reuse: Architecture, process, and organization for business success. In *ICCSSE '97: Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, page 86, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8135-7.

[50] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24 (11):43–75, 1989.

[51] Integral Systems. EPOCH T&C Directives and STOL Functions Reference Manual. http://www.integ.com/, 2000.

[52] G. Booch J., Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Professional, 2005. ISBN 0321267974.

[53] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1-2. Springer-Verlag, 1992.

[54] J. P. Katoenr. Labelled transition systems. Lecture Notes in Computer Science, pages 615–616. Springer Berlin / Heidelberg, 2005.

[55] M. Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference, 4th Edition*. Wrox, 2008. ISBN 978-0-470-19274-0.

[56] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for Grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005. ISSN 1049-331X.

[57] P. Klint, T. van der Storm, and J. Vinju. RASCAL: A domain-specific language for source code analysis and manipulation. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:168–177, 2009.

[58] D. E. Knuth. The genesis of attribute grammars. In *WAGA: Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-53101-7.

[59] A. Koller, M. Regneri, and S. Thater. Regular tree grammars as a formalism for scope underspecification. In *Proceedings of the 46th ACL-Conference on Human Language Technologies*, pages 218–226, Columbus, Ohio, USA, 2008. Association for Computational Linguistics.

[60] J. Kort, R. Lämmel, and C.Verhoef. The Grammar Deployment Kit. *Electronic Notes in Theoretical Computer Science*, 65, 2002.

[61] J. Krein, A. MacLean, D. Delorey, D. Eggett, and C. Knutson. Language entropy: A metric for characterization of author programming language distribution. In *Fourth International Workshop on Public Data about Software Development (WoPDaSD '09)*, page 6, June 2009.

[62] I. Kurtev. State of the art of QVT: A model transformation language standard. *Applications of Graph Transformations with Industrial Relevance*, pages 377–393, 2008.

[63] R. Lämmel. Grammar Testing. In *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[64] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[65] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, 2001.

[66] R. Lämmel and V. Zaytsev. An introduction to grammar convergence. In *IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods*, Lecture Notes in Computer Science, pages 246–260, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00254-0.

[67] R. Lämmel and V. Zaytsev. Recovering grammar relationships for the Java language specification. *IEEE International Workshop on Source Code Analysis and Manipulation*, 0:178–186, 2009.

[68] M. Lawley and J. Steel. Practical declarative model transformation with TEFKAT. *Satellite Events at the MoDELS 2005 Conference*, pages 139–150, 2006.

[69] P. Leinonen. Automating XML document structure transformations. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 26–28, New York, NY, USA, 2003. ACM. ISBN 1-58113-724-9.

[70] P. M. Lewis and R. E. Stearns. Syntax-directed transduction. *J. ACM*, 15(3):465–488, 1968. ISSN 0004-5411.

[71] D. Liu and D. Gildea. Improved tree-to-string transducer for machine translation. In *StatMT '08: Proceedings of the Third Workshop on Statistical Machine Translation*, pages 62–69, Morristown, NJ, USA, 2008. Association for Computational Linguistics. ISBN 978-1-932432-09-1.

[72] N. Lobo, T. Kasparis, M. Georgiopoulos, F. Roli, J. Kwok, G. C. Anagnostopoulos, and M. Loog. Structural, syntactic, and statistical pattern recognition. In *Vision, Pattern Recognition, and Graphics*. Springer Publishing Company, Incorporated, 2008. ISBN 3540896880, 9783540896883.

[73] I. A. Mason and C. L. Talcott. Program transformation via contextual assertions. In *Logic, Language and Computation*, pages 225–254, 1994.

[74] M. W. Matlin. *Cognition, 6th Edition*. John Wiley & Sons, Inc., 2005. ISBN 978-0-471-45007-8.

[75] C. Matthiessen, A. Caffarel, and J. R. Martin. *Language typology : a functional perspective*. John Benjamins, Amsterdam ; Philadelphia :, 2004. ISBN 1588115593 9027247668.

[76] N. Mecredy and A. Armitage. Herschel Planck central checkout system system user manual. http://www.terma.com/, 2004.

[77] T. Mens, K. Czarnecki, and P. Van Gorp. A taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[78] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[79] P. Mosses. Component-based description of programming languages. In *Visions of Computer Science - BCS International Academic Conference*, 2008.

[80] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[81] M. J. Nederhof and G. Satta. Kullback-Leibler distance between probabilistic context-free grammars and probabilistic finite automata. In *COLING '04: Proceedings of the 20th international conference on Computational Linguistics*, page 71, Morristown, NJ, USA, 2004. Association for Computational Linguistics.

[82] Object Management Group. Meta object facility (MOF) core specification version 2.0. http://www.omg.org/spec/MOF/, 2006.

[83] D. Ordóñez Camacho. Towards a language-independent intentional views framework. Université catholique de Louvain, 2004.

[84] D. Ordóñez Camacho and K. Mens. Appareil: A tool for building automated program translators using annotated grammars. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 489–490. IEEE, 2008.

[85] D. Ordóñez Camacho and K. Mens. Using annotated grammars for the automated generation of program transformers. In *Ingénierie Dirigée par les Modéles, IDM2007, proceedings*, pages 7 – 24, Toulouse, France, 2007. Eds. Antoine Beungard & Marc Pantel. ISBN 978-2-7261-1292-7.

[86] D. Ordóñez Camacho, K. Mens, M. van den Brand, and J. Vinju. Automated derivation of translators from annotated grammars. *Electronic Notes in Theoretical Computer Science*, 164, Issue 2:121–137, 2006.

[87] D. Ordóñez Camacho, K. Mens, D. Quigley, and J. Cater. Issues and problems in tests and operations languages translation. *Proceedings of the SpaceOps 2008 Conference*, 2008.

[88] D. Ordóñez Camacho, K. Mens, M. van den Brand, and J. Vinju. Automated generation of program translation and verification tools using annotated grammars. *Science of Computer Programming*, 75 (1-2):3–20, 2010. ISSN 0167-6423.

[89] S. Y. Park and S. D. Kim. A systematic method for scoping core assets in product line engineering. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 491–498, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2465-6.

[90] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, May 2007. ISBN 0978739256.

[91] L. Petrone. On the use of syntax-based translators for symbolic and algebraic manipulation. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 224 – 237, 1971.

[92] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, APPSEM 2000*, Lecture Notes in Computer Science, pages 378–412, London, UK, 2002. Springer-Verlag. ISBN 3-540-44044-5.

[93] J. Podani. *Introduction to the Exploration of Multivariate Biological Data*. Backhuys Publishers, Leiden, The Netherlands, 2000. ISBN 90-5782-067-6.

[94] R. I. Podlovchenko. Models of sequential programs used to study functional equivalence of programs. *Cybernetics and Systems Analysis*, 15 (1):22–31, 1979.

[95] D. Quigley and S. J. Cater. Satellite test and operation procedures cost reduction through standardization. *IEEE Aerospace Conference*, page 10, 2006.

[96] D. Quigley and A. Monham. Mission operations preparation management: An effective end-to-end approach. *IEEE Aerospace Conference*, pages 3846 – 3855, 2004.

[97] D. J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[98] C. R. Rao. Diversity and dissimilarity coefficients: A unified approach. *Theoretical Population Biology*, 21(1):24 – 43, 1982. ISSN 0040-5809.

[99] J. L. Roca. An entropy-based method for computing software structural complexity. *Microelectronics and Reliability*, 36(5):609 – 620, 1996. ISSN 0026-2714.

[100] K. Schmid. Scoping software product lines: an analysis of an emerging technology. In *Proceedings of the first conference on Software product lines : experience and research directions*, pages 513–532, Norwell, MA, USA, 2000. Kluwer Academic Publishers. ISBN 0-79237-940-3.

[101] A. Schurr. PROGRES, a visual language and environment for PROgramming with Graph REwrite Systems, 1994.

[102] A. Schurr. Specification of graph translators with triple graph grammars. *Proceedings of the 20 International Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany, June 1994*, 1994.

[103] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949.

[104] K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe. IEEE*, pages 1310 – 1315. IEEE Computer Society, 2005.

[105] C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, 1995.

[106] J. E. Simpson. *XPath and XPointer: Locating Content in XML Documents*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. ISBN 0596002912.

[107] E. E. Smith and D. L. Medin. *Categories and concepts*. Harvard University Press, Cambridge, Mass., 1981. ISBN 0674102754, 0674157257.

[108] C. Stirling. The joys of bisimulation. In *MFCS*, volume 1450 of *Lecture Notes in Computer Science*, pages 142–151. Springer, 1998. ISBN 3-540-64827-5.

[109] T. Sturm, J. von Voss, and M. Boger. Generating code from UML with Velocity templates. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 150–161, London, UK, 2002. Springer-Verlag. ISBN 3-540-44254-5.

[110] Integral Systems. STOL programmer's reference manual. http://www.integ.com/, 2000. Integral Systems Inc. Lanham, Maryland, USA.

[111] J. Levine T., Mason D., and Brown. *LEX & YACC, 2nd Edition (A Nutshell Handbook)*. O'Reilly, October 1992. ISBN 1565920007.

[112] G. Taentzer, D. Müller, and T. Mens. Specifying domain-specific refactorings for AndroMDA based on graph transformation. In *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007*, pages 104–119, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89019-5.

[113] A. A. Terekhov. Automating language conversion: a case study. In *IEEE International Conference on Software Maintenance*, pages 654–658. IEEE Computer Society Press, November 2001.

[114] A. A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, November/December 2000.

[115] L. J. Timmermans, T. Zwartbol, B. A. Oving, and A. A. Casteleijn. From simulations to operations: Developments in test and verification equipment for spacecraft. *DAta Systems In Aerospace, DASIA*, 2001.

[116] S. Unger and F. Mueller. Handling irreducible loops: optimized node splitting versus DJ-graphs. *ACM Trans. Program. Lang. Syst.*, 24(4): 299–333, 2002. ISSN 0164-0925.

[117] M. van den Brand and P. Klint. ASF+SDF Meta-Environment user manual. http://www.meta-environment.org/, 2005.

[118] M. van den Brand, A. van Deursen, J. Heering, H. de Jonge, M. de Jonge., T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[119] F. van der Linden and H. Obbink. ESAPS - engineering software architectures, processes and platforms for system families. *Software Architectures for Product Families*, pages 244–252, 2000.

[120] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000. ISSN 0362-1340.

[121] N. Veerman. Towards lightweight checks for mass maintenance transformations. *Science of Computer Programming*, 57(2):129–163, 2005.

[122] N. Veerman. Restructuring Cobol systems using automatic transformations, November 2001.

[123] S. D. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. *Lecture Notes in Computer Science*, 5301:630–644, September 2008. ISSN 0302-9743. In K. Czarnecki and I. Ober and J.-M. Bruel and A. Uhl and M. Voelter (eds.) Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008). Toulouse, France, October 2008.

[124] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[125] E. Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[126] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[127] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl (3rd Edition)*. O'Reilly, 2000. ISBN 0596000278.

[128] P. Walmsley. *Definitive XML Schema*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130655678.

[129] D. M. Weiss and R. Lai Chi Tau. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-69438-7.

[130] L. J. Whaley. *Introduction to Typology. The Unity and Diversity of Language.* Sage Publications, Inc., 1997. ISBN 9780803959637.

[131] M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger. A petri net based debugging environment for QVT relations. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE 2009)*, pages 1–12. IEEE, 2009.

[132] D. M. Yellin. *Attribute grammar inversion and source-to-source translation.* Springer-Verlag New York, Inc., New York, NY, USA, 1988. ISBN 0-387-19072-4.