LDTA 2006 Preliminary Version

Automated Derivation of Translators From Annotated Grammars

Diego Ordonez Camacho [a,1,4] Kim Mens [a,1]
Mark van den Brand [c,3] Jurgen Vinju [b,2]

[a] *Department of Computer Science, Université catholique de Louvain, Louvain-la-Neuve, Belgium*

[b] *Department of Software Engineering, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands*

[c] *Department of Mathematics and Computer Science, Technical University Eindhoven, Eindhoven, The Netherlands*

Abstract

In this paper we propose a technique to automate the process of building translators between operations languages, a family of DSLs used to program satellite operations procedures. We exploit the similarities between those languages to semi-automatically build a transformation schema between them, through the use of annotated grammars. To improve the overall translation process even more, reducing its complexity, we also propose an intermediate representation common to all operations languages. We validate our approach by semi-automatically deriving translators between some operations languages, using a prototype tool which we implemented for that purpose.

Key words: operations languages, language translation, language transformation, automatic translation, grammarware

1 Introduction

1.1 Motivation

As opposed to general-purpose programming languages, which are designed to solve computing problems in any domain, *domain-specific languages* (DSLs)

[1] {Kim.Mens | diego.ordonez}@uclouvain.be
[2] Jurgen.Vinju@cwi.nl
[3] m.g.j.v.d.brand@tue.nl
[4] This research was funded by the Wallonian Region in Belgium and the European Social Fund, in the context of the FIRST Europe Objectif 3 research project 415748 entitled "Une approche paramétrique de réingénierie logicielle — APPAREIL".

are typically smaller programming languages dedicated to a specific task or application domain [21]. An example of such a specific application domain is that of *spacecraft mission planning*, where spacecrafts receive commands from so-called operators. These commands are described in DSLs called *operations languages* (OLs). OLs have been designed with the purpose of regrouping the commands sent to a spacecraft into operations, which are specialized programs that describe an organised procedure for a spacecraft.

In the domain of spacecraft mission planning there exist probably as many OLs as there are spacecraft operators. These languages can have very different syntaxes and language constructs. Nevertheless, since they all have the same goal and respect known standards on satellite construction and operation, all of them share many features. More precisely, all OLs share a common semantical foundation and programming paradigm: they are all imperative and flow-driven languages.

In an attempt to make the design and testing of spacecraft procedures easier, many operators use specialized software applications. Designers and implementers of such applications are confronted with the need of making them generic, so that they can be employed by as many operators as possible, regardless of the actual operations language they prefer to use. Although these applications already allow operators to design and edit procedures in any OL, they lack the ability to translate these procedures between OLs. In addition, these applications should be easily extendable to support new operations languages.

1.2 Approach

The approach we propose in this paper is a generic technique to semi-automatically derive translators from one OL to another, based on the corresponding context-free grammars of those languages annotated with extra information at the production and non-terminal level.

The proposed technique does more than providing an alternative solution to the old problem of language translation. It also helps reducing development time of a rather time-consuming part of the process of building program translators. Furthermore, the modularity of our technique enables future reuse of translation modules, when writing or deriving new translators for other languages.

We implemented a prototype of an algorithm that semi-automatically derives translators, by using ASF+SDF [4,11] and the ASF+SDF Meta-Environment [5].

In summary, the main contributions of our technique are :

(i) a mechanism that automates the process of building translators between different operations languages, based on the ideas of *grammarware* development [12];

(ii) a common intermediate representation for all operations languages;

(iii) a prototype implementation of the derivation tool that could be incorporated as a library into the Asf+Sdf Meta-Environment.

Although we validate and illustrate our approach and algorithm only on the case of operations languages, there exist other families of languages that have a common semantical foundation, e.g. databases design languages or query languages. We conjecture that our technique could be applied in such domains too.

The remainder of this paper is structured as follows. In Section 2, we analyse the research problem in more detail and take a closer look at the domain of satellite missions and procedures. The annotated grammars technique, our solution to the research problem, is explained in detail in Section 3, and validated on the case of operations languages. We introduce our common intermediate representation for OLs in Section 4. Finally, in Sections 5 and 6 we present the results of the first experiments performed with our approach, highlight advantages and shortcomings of our technique, and summarize our contributions.

2 Context

2.1 Operations languages

Spacecraft mission operations are all activities related to the planning, execution and control of satellite behavior. One major element of mission operations is the flight operations plan which contains all information required to execute the operations, including all flight control procedures and contingency recovery procedures. A procedure is the specified way to perform an activity, and is the principal mechanism employed by the end-user to control the space system during the execution of an operation. These procedures are written, depending on the mission control center that operates the satellite, using one among the multitude of operations languages that exist.

As an example, Figure 1 shows part of a test procedure written in the Pluto [10] language. Pluto supports instructions that can be found in many other languages, like control flow statements (while, if), variable assignments and logging. It also supports dedicated instructions, provided by most OLs, to communicate directly with the satellite. Examples of the latter are the instructions `Get_Engineering_Value of DHT30100` at line 5 and `initiate and confirm PHC10117` at line 11.

This similarity in instructions and semantics among OLs makes it feasible to translate from one to another in a highly automated way (even though the problem of automatically translating from any language to any other is, in general, unsolvable).

```
log "PROCEDURE PlutoTest_45_03 Step_1";
relVAR := 3600 sec;
bootNotRealised := TRUE;
while (bootNotRealised AND relVAR > 0 sec ) do
   bootNotRealised := ((Get_Engineering_Value of DHT30100)=ACTIVE);
   if (bootNotRealised) then
   wait 1 sec; relVAR := relVAR - 1 sec ;
   end if;
end while;
if bootNotRealised then
   initiate and confirm PHC10117;
end if
```

Fig. 1. Code fragment of a test procedure in the Pluto operations language.

2.2 The research problem

This research addresses two related problems. One is the classical problem of generic language translation, which is still under active investigation [14,19,20]. A second problem is, when defining translators between many different languages in a same family of languages, many of the translators will have similar fragments. To avoid having this repetition a modular translation technique is beneficial.

To address the problem of translating between arbitrary OLs, providing a specific translator for every source and target language combination would obviously lead to a combinatorial explosion of translators. An alternative approach — that is part of our final solution — is to introduce an additional language that can act as intermediate representation when translating between any two OLs. We need to design this intermediate representation in such a way that it allows to reuse language and transformation components, in order to decrease the manual effort when adding additional languages to our set of translators.

But even when passing via such an intermediate representation, the core of our translation problem remains. Although it reduces significantly the number of translators that need to be implemented, we still need to build an important amount of them. Taking into account the fact that all languages in our domain share many features, we hypothesize that the translators themselves are also similar to a large extent, and that we can exploit this similarity to automate the process of building them.

This similarity in the translators was confirmed by an experiment, where we programmed a set of translators by hand. During that experiment we observed that in many of the translators certain coding patterns appeared over and over again. It was precisely this repetition that we wanted to exploit to further automate the process of building language translators between any two OLs.

2.3 Our solution in a nutshell

Our solution to the automated translation of procedures between multiple OLs is composed of the following steps, each of which is explained in more detail in the subsequent sections.

(i) We automate the process of building program translators between two OLs, by taking advantage of language similarities. We map source to target languages by annotating their grammars, and we provide these annotated grammars to our system, which then produces an automatic translator. This automatic translator is built in a modular way and can easily be extended with further transformation rules to complete the translator.

(ii) We design an intermediate representation common to every OL. Like this we can translate from any of these languages to this representation, as well as from the intermediate representation to any such language. This intermediate representation provides a generic syntactic and semantic model for the family of OLs, in terms of which to define translators for languages in that family.

3 Annotated Grammars

Syntax-directed translation [1] 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 [15] that specifies the input-output relation of the translation and deduces the actual translator from that relation.

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, taking as input the grammars of both languages, previously annotated with constructor and label information to establish a mapping [16] between corresponding language constructs. The mechanism provides a way to automatically generate the translator for some of the productions in the grammars, as well as basic support to extend that translator with the necessary transformations for the remaining productions.

Although many existing tools could be used to implement this solution, as for instance [3,7,9,23], we have chosen Asf+Sdf and the Asf+Sdf Meta-Environment for implementing our prototype. Asf+Sdf 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 [6] 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. We discuss SDF in more detail in Section 3.1, followed by a brief summary of ASF in Section 3.2.

3.1 SDF

The Syntax Definition Formalism is a formalism for the definition of grammars, that combines completely lexical and context-free syntax definition. It supports arbitrary context-free syntax, because of the underlying generalized parsing algorithm, and provides several disambiguation methods to deal with ambiguous grammars. It also supports modularization and reuse of syntax definitions [22].

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 decorated 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:

$$A\ B\ C \rightarrow X\{\ldots, \mathbf{cons}(ConstructorName), \ldots\}$$

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:

$$\mathbf{label_a}\colon A\ \mathbf{label_b}\colon B\ \mathbf{label_c}\colon C \rightarrow X\{\ldots, \mathit{cons}(ConstructorName), \ldots\}$$

This last feature is specially handy to avoid certain mapping problems when, for instance, matching non-terminals in source and target productions do not appear in the same order.

3.2 ASF

ASF is a formalism for defining conditional rewrite rules. These rewrite rules can be used to define a semantics, for a language specified in the SDF part, 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 of the variables to the matched subterms in the concrete syntax tree. See [6] for a more detailed description.

Figure 2 shows the context-free syntax rules for two different occurrences of conditional language constructs (i.e., an if statement and a conditional evaluation), and the rewrite function `f` for mapping one of the language constructs

```
context-free syntax
  "if" Expr "then" StatsS "fi"      -> IfS
  "eval(" Expr "," BlockT ")"       -> EvalT
context-free syntax
  f(IfS)                            -> EvalT
variables
  "$Expr$"                          -> Expr
  "$StatsS$"                        -> StatsS
equations
  f(if $Expr$ then $StatsS$ fi) =   eval( $Expr$ , f($StatsS$) )
```

Fig. 2. An example of a simple translation function expressed in ASF+SDF.

```
    If EvalBlock                 eval( Condition ) {
    then TrueBlock               iftrue( StatementList )
    else FalseBlock              otherwise( StatementList ) }
```

Fig. 3. Conditional constructs in two different languages.

to the other. It illustrates the unconditional rewrite rules in ASF as well as the use of variables.

3.3 Preliminary Experiment

During a preliminary experiment, eventually leading to the work presented in this paper, we manually built translators from the operations languages Pluto and UCL [2] to and from an intermediate representation language IRL, explained in more detail in Section 4. We started with a subset of constructs for these languages, consisting mainly of control flow structures, and programmed four translators: Pluto to IRL, IRL to Pluto, UCL to IRL, and IRL to UCL.

The sum of the number of ASF transformations we had to implement for the four translators was 91, but the implementation of 73 of these transformations (about 80%) followed a repeatable pattern. It was like the rewriting rules were acting as a bridge between source and target grammars, with an almost one-to-one correspondence between productions and non-terminals. Only 18 of all the transformations (slightly less than 20%) were "non-trivial", requiring more knowledge than that could be deduced from the grammar. This observation led us to the solution proposed in Section 3.4.

3.4 Grammar Annotations

Now that we have explained all preliminaries, let us return to the core of the problem, which is to provide automated support for building source-to-source translators for operations languages. Since these languages belong to the same family, they have many commonalities, and thus the translators involve a lot of trivial transformations that could be generated automatically.

For example, the language constructs shown in Figure 3 belong to two

EvalBlock	⇔	*Condition*
{*TrueBlock*, *FalseBlock*}	⇔	*StatementList*

Fig. 4. Equivalent non-terminals in Figure 3.

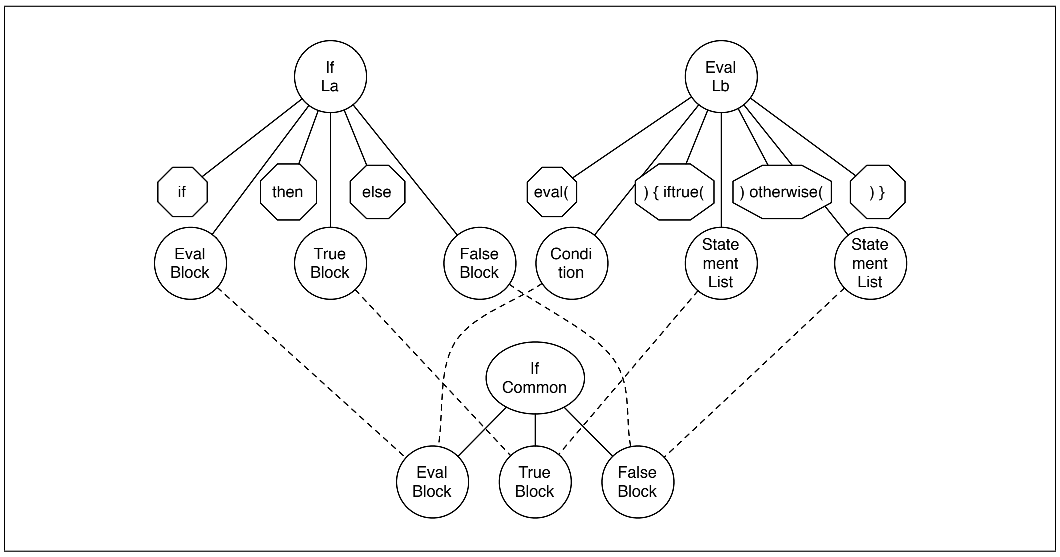

Fig. 5. Abstraction of a common construct.

```
"if" EvalBlock                "eval(" Condition ")" "{"
"then" TrueBlock              "iftrue(" StatementList ")"
"else" FalseBlock             "otherwise(" StatementList ")"
-> If                         "}" -> Eval
```

Fig. 6. The SDF rules for the two different conditionals

different languages. Although the syntactic structure of both differ, both constructs have the same semantics: they evaluate a boolean condition, and depending on its truth value, they execute one of the statement blocks. In this example it is easy to establish that the productions for this construct are equivalent, as well as they are their non-terminals, like in Figure 4.

As Figure 5 illustrates, such an equivalence can be regarded as an 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. Like this we build a bridge between the two languages, allowing us to translate specific instances of a construct in one language to its counterpart in the other language. Figure 6 shows the SDF productions for the language constructs of Figure 3.

There are equivalences in the left-hand sides of these SDF productions as well. For simple cases no additional work should be necessary, because once all productions are matched, often the system can infer how non-terminals occurring in the left-hand side of both productions can be matched as well. The mere order in which they appear could be enough to establish a one-to-one mapping. However, there can be many exceptions to this general rule: different number of non-terminals in both productions, different order, more than one non-terminal of the same type, and so on; a more accurate solution is needed.

```
"if" cond:EvalBlock         "eval(" cond:Condition ")" "{"
"then" trueb:TrueBlock      "iftrue(" trueb:StatementList ")"
"else" falseb:FalseBlock    "otherwise(" falseb:StatementList ")"
-> If {cons("If")}          "}" -> Eval {cons("If")}
```

Fig. 7. Two annotated equivalent productions

To address this problem we associate labels to every non-terminal in the left-hand side of a production. Figure 7 gives an example of two fully annotated equivalent productions. The resulting AST: $If(cond, trueb, falseb)$, is equivalent for both productions although they belong to different languages.

3.5 Summary of the approach

In summary, to derive a translator with our approach, these are the basic steps to follow:

(i) Analyze the grammars and look for productions with the same meaning. This is a manual process that requires good knowledge of both languages and trusts on the user entirely to match the grammars, thus establishing the semantic mapping or bridge.

(ii) For every production S in the *Source* language, find the production T in the *Target* language that fulfills the equivalence requirements, and annotate both productions with constructor information.

(iii) Link the left-hand sides of both productions. For every couple of non-terminals $[A, X]$ having A in production S (in *Source*), and X in production T (in *Target*), where A is equivalent to X, label both non-terminals with the same attribute name.

(iv) Continue this process until every possible equivalence between productions and non-terminals is defined.

(v) Feed the system with the annotated grammars and as a result an Asf+Sdf translator system from *Source* to *Target* will be returned.

(vi) Manually treat those cases where mappings could not be derived automatically, mainly by adding transformations to the translator.

3.6 Transformation Example

We now illustrate the approach by deriving a translator for the two languages shown in Figures 8 and 9. Note that, in those figures, we already performed steps (i) to (iv) of our approach, so the grammars have already been annotated by the user with constructor information and labels.

```
module Source
imports Expr
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")}
```

Fig. 8. Part of the source grammar

```
module Target
imports Expr
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")}
```

Fig. 9. Part of the target grammar

```
module Expr
context-free syntax
"true" | "false" | "nil" | "nil2"              -> Expr
"not" Expr                                     -> Expr
```

Fig. 10. Part of the common grammar

The transformation system starts by relating productions in the source and target grammars with 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. E.g., for the productions with constructor attribute `cons("IfThen")`, a translation function `f(IfS) -> EvalT` will be derived.

The rewrite equations for the transformation system will now be generated based on the left-hand sides of both productions. The translation function `f(IfS) -> EvalT` will be expressed like
`f(if $Expr$ then $StatsS$ fi) = 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, except for non-terminals like *Expr*, that thanks to languages similarities and environment modularization, are imported by both the input and output grammars — this common grammar is shown in Figure 10.

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

Fig. 11. Signature of 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($StatS*$)
```

Fig. 12. Equations of translation functions

```
From:                                To:
proc                                 start(
if true then nil else nil2 fi        eval(true, nil)
                                     eval(not true, nil2)
while true do nil nil2 nil od        loop(true, nil nil2 nil)
endproc                              )
```

Fig. 13. Translation example

For the grammars of Figures 8 and 9, the signature of the translation functions (Figure 11) and the actual translation equations (Figure 12) are generated automatically.

3.7 Example of Manual Intervention

Finally, we illustrate how to handle those cases where we fail to establish a mapping between productions. Whenever that happens, extra transformations need to be added manually to the automatically derived translator.

For example, the production with constructor attribute `"IfThenE"` in Figure 8 has no equivalent in the target grammar of Figure 9. Manual intervention is needed to allow the translator to handle this language construct. A possible solution for this particular example is:

(i) We modify the translation function for `IfS` by changing the cardinality of the resulting type: `f(IfS) -> EvalT+`

(ii) And we 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 have obtained a complete translator that can translate any program in *Source* to *Target*. For instance, the program in the left column of Figure 13 gets translated to the one on the right.

4 Intermediate Representation Language (IRL)

Even though we now have an automated mechanism for deriving source-code translators between any two operations languages, we still have a combinatorial explosion of possible translators if we want to translate from any language to any other language in that family. To address that problem, as announced in Section 2.2, we designed an Intermediate Representation Language (IRL), that abstracts the behavior of all languages in our family of operations languages, and provide translators only for each of the languages to and from the IRL. As such, we only need to build $2n$ translators (instead of $n(n-1)$), where n is the number of OLs, and adding a new language to the set requires adding only 2 extra translators (as opposed to $2n$).

To design our IRL we selected a representative sample of OLs like Pluto[5], UCL or Stol [18]. However, we were a bit hindered in our work because for some OLs no documentation describing their complete grammar and semantics is available. In addition, due to language incompatibilities, in some cases abstracting the commonalities among grammars may lead to a loss of information. For instance, since only one of the OLs allows to associate a name to every “block of instructions”, this information is not put in the common grammar and thus will be lost.

Based on the language constructs encountered in Figure 1, for example, we may decide to include the following constructs in our IRL: *Log, Assignment, Loop, If, GetValue, InitiateCommand*. The part of our IRL description for the *If* and *Loop* constructs may look as presented in Figure 14. Notice that in our SDF representation we already make use of labels and constructs, providing additional semantic information. The IRL has an XML-like syntax.

We regard our IRL as an evolving system. For its initial design, we considered a representative set of languages, and commonalities were derived from this set. However, whenever we want to add another OL to this “system” we may discover constructs other than those already considered. To deal with such constructs we designed the IRL in a layered way, as shown in Figure 15. Language constructs common to most OLs belong to a *Core* module. Surrounding that module we have an additional layer of *Extensions*, where we can add constructs that are shared by some languages but that are not general enough to merit being part of the core.

For instance, since not all OLs provide a *For* loop, we prefer to add this construct as an extension to the IRL, but not to the core. This extension can still be reused by all OLs that provide such a construct. Together with a production describing this language construct as an extension to the IRL, we provide a transformation from that extension to the core layer of the IRL: *Ext1* or *Ext2* to *Core*. This transformation will be a rewriting rule as explained in more detail in Section 3.2.

[5] As one of the goals of Pluto is to become the future standard for OLs, it is a very representative language to consider.

```
%% If statement
"<if-step>"
    if:M-Ifonly
    else:M-Else?
"</if-step>"
        -> M-If {cons("If")}

%% Generic Loop statement
"<loop>"
    "<checkbefore/>" | "<checkafter/>"
    "<loopiftrue/>" | "<loopiffalse/>"
    cond:M-Expression
    block:M-Block
"</loop>"
        -> M-Loop {cons("Loop")}
```

Fig. 14. Fragment of the language definition of the Intermediate Representation Language for the family of OLs

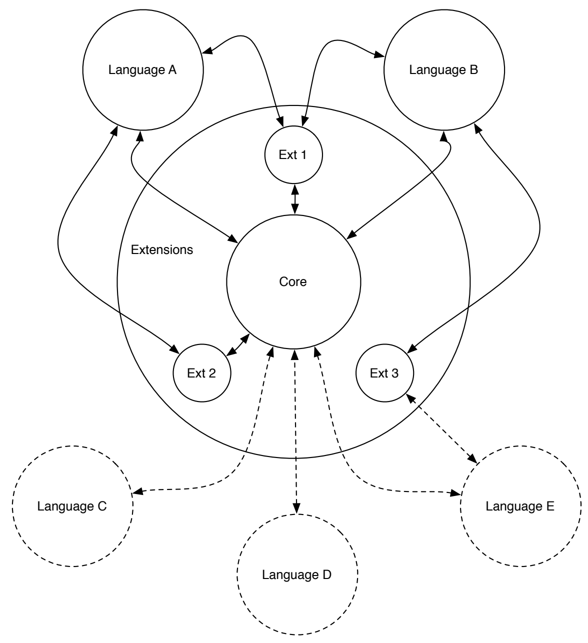

Fig. 15. Intermediate Representation Language structure

There can be cases where no possible transformation exist to go from an extension to the *Core* — as illustrated by *Ext3* — maybe because it is too specific to certain languages or implementations (e.g., threading or exception handling). These "unlinked" extensions will have to be managed as exceptional cases, only shared by a subset of the languages and, therefore, not fully generalizable.

The basic idea behind the IRL structure is to obtain reuse through modularization. For every language construct present in an extension module, we

```
%% While statement

"<while>"
    cond:M-Expression
    block:M-Block
"</while>"
       -> M-While {cons("While")}
```

Fig. 16. A “while” extension inside the IRL.

```
%% While

<loop>
    <checkbefore/>
    <loopiftrue/>
    ...an expression...
    ...a block...
</loop>"
```

Fig. 17. A “while” instance in the core IRL.

provide the syntax of the productions and a semantics by mapping it to more primitive constructs in the core module. This mapping typically needs to be implemented by hand; however, once an extension has been defined, it can be used directly by additional languages implementing the same construct. To illustrate these ideas, below we give some concrete translation examples that illustrate the flexibility of the IRL and how to extend it with new constructs providing straightforward mappings to the end user, while preserving generality.

Let us revisit the “generic loop statement” in Figure 14. This is a generic construct that can express different types of loops (e.g., while-do, do-until) by using the additional terminal symbols to specify the desired semantics of a particular type of loop. For instance, by choosing the non-terminals <`checkbefore`/> and <`loopiftrue`/> we can express that we want a typical while loop, where the condition is checked before the statement block is executed, and the loop continues only when the condition evaluates to true.

The generality of such a compact loop construct also has some drawbacks. It may make particular translators that use this generic construct, for example, to express a particular type of loop, more difficult to understand than when a more concrete construct would have been present in the IRL. But nothing prohibits us from offering such more concrete constructs (together with their mapping to the more generic construct in the IRL) as extensions to the IRL. In such an extension, a while construct could for example be expressed more directly and naturally as shown in Figure 16. This extension would then transform automatically to the generic loop construct in the core IRL, producing a structure as in Figure 17.

```
%% Do - Until statement

"<do-until>"
    block:M-Block
    cond:M-Expression
"</do-until>"
       -> M-DoUntil {cons("Until")}
```

Fig. 18. A “do-until” extension inside the IRL.

```
%% (A)                             %% (B)
<while>                            <loop>
    A-Bool-Cond                        <checkbefore/>
    A-Stats-Block                      <loopiftrue/>
</while>                               A-Bool-Cond
                                       A-Stats-Block
                                   </loop>

%% (C)                             %% (D)
<if-step>                          <if-step>
    A-Bool-Cond                        A-Bool-Cond
    <loop>                             <do-until>
       <checkafter/>                      A-Stats-Block
       <loopiffalse/>                     <not/> A-Bool-Cond
       <not/> A-Bool-Cond              </do-until>
       A-Stats-Block               </if-step>
    </loop>
</if-step>
```

Fig. 19. A (simplified) chain of transformations (from A to D) performing a loop inversion inside the IRL.

Now, let us consider a slightly more complex situation, where we would want to translate from some language A that provides only “do-until” loops, to a language B that provides only “while” loops, by passing via the IRL. First of all, as we already explained for the “while” construct, we would need an extension like the one in Figure 18, that knows how to translate “do-until” statements to the generic loop construct in the IRL. Secondly, we need a transformation scheme from such “do-until” statements in language A to “while” statements in language B, as illustrated in Figure 19. (In this particular case, this requires a loop inversion.)

It is true that the rewrite rules for this chain of transformations needs to be written by hand, but once that has been done they can readily be reused for translating between other languages that have similar constructs.

5 Discussion

One of the obvious limitations of our approach, as explained in Section 3, is that the deduction of language translators is not fully automatic. Manual intervention is needed at the start of the process, to annotate the grammars, instructing the deduction algorithm how to map constructs. This user intervention, however, is no additional work. Even when manually programming a translator, a deep understanding of how corresponding constructs in two languages relate, would be required. In our approach we are just stating these relations explicitly, to automate further steps. Another manual intervention is needed at the end of the process, to extend the produced translator(s) with extra transformation rules for those constructs where no initial mapping could be provided.

Another issue is that, in order to make it easier to map the grammar of one language to another, it is important that they have a similar structure. In our case, we didn't really suffer from this problem because, for each of the languages we experimented with, we first designed the grammars for those languages by hand, based on information from the language manuals and documentation. This naturally led to a set of grammars that were structured in a very similar way. If grammars for those languages would already have been available, however, it would have made sense to first perform a normalization step, as suggested by [13], to bring the different grammars in a similar form.

The more similar the languages are, the more the process of deducing a translator between such languages can be automated. We conducted some experiments with lightweight versions of both Pluto and our IRL, and observed that our approach was highly automatic, being able to deduce most of the transformation rules to translate from one language to another, without the need of any human intervention at the end. The few cases where mappings between language constructs could not be defined straightforwardly, often could be solved by simple grammar manipulations (adding or removing extra non-terminals) to make the grammars more similar, thus avoiding the manual intervention at the end.

More specifically, we achieved good results transforming between command executions, objects definitions, flow control structures and expressions. All of these constructs, however, are very local, not needing more information than provided by the productions themselves. Dealing with more global constructs like goto-statements, or passing from untyped representations to typed ones, cannot be accomplished with our simple translation schema, and would require more complex transformations rules to be programmed by hand.

Finally, our approach could be seen as too focused on syntax, which is partially true because our particular problem (translating between operations languages) is mostly syntactic. But even in those cases where the problem would be more semantic, syntax would need to be taken into account as well, and our approach could be considered at least for that aspect. One could also argue that only trivial translations can be achieved with our technique, but thanks to the environment we have chosen and the design of our intermediate representation, we can easily add more complex transformations — as we have illustrated in Section 4 — which can be reused later on in other translators with no additional programming effort.

5.1 Related work

A lot of related work exists in the domain of language translation, and it is not our intention to present an exhaustive survey of the field here. We just present a few other interesting approaches that are closely related or complementary to ours. In [26] multi-language translation is tackled through a minimal central representation, and a restricted form of invertible grammars. An expansion mechanism is proposed in [25] for modularly adding new features to a language, using attribute grammars. Graph translators are studied in [17] where relationships are described through additional correspondence rules. Finally, [24] provides an alternative way to generate translators based on syntax-directed rules sets.

5.2 Future work

As our work has a strong practical objective, the next logical step is to turn our prototype into a production-level tool, that can be incorporated in industrial tools such as those mentioned in subsection 1.1.

Even though current experimentation has been performed only in the domain of operations languages, we believe the approach is generic enough to be used in many other domains as well. To validate this claim, further experimentation will be performed to confront our approach with languages in other domains (e.g. the database domain as in [8]).

6 Conclusions

We have shown how annotated grammar definitions can support automated generation of translators between languages. Although we have used the family of operations languages as a case study throughout this paper, we believe that our technique would be helpful for other domain-specific language families as well, especially when dealing with intensive translation of programs between multiple representations having very similar semantics.

We have also shown, using the family of operations languages as an example, how an intermediate representation structure can provide an extensible, modular and reusable "translation system". Finally, we pointed out some specific advantages and disadvantages of our technique, and suggested some interesting avenues for future work in this field.

Acknowledgement

We thank Paul Klint, Darius Blasband and Rob Economopoulos for having proofread and commented on earlier versions of this paper.

References

[1] Aho, A. V. and J. D. Ullman, *Translations on a context free grammar*, in: *STOC '69: Proceedings of the first annual ACM symposium on Theory of computing* (1969), pp. 93–112.

[2] ASTRIUM, *User control language reference manual* (2003).

[3] Baxter, I. D., *Dms: program transformations for practical scalable software evolution*, in: *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution* (2002), pp. 48–51.

[4] Bergstra, J. A., J. Heering and P. Klint, "Algebraic specification," ACM Press, New York, NY, USA, 1989.

[5] Brand, M. v. d., A. v. Deursen, J. Heering, H. d. 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)*, LNCS **2027** (2001), pp. 365–370.

[6] Brand, M. v. d. and P. Klint, *ASF+SDF Meta-Environment user manual* (2005).

[7] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. Quesada, "Maude: Specification and Programming in Rewriting Logic," SRI International (1999).

[8] Cleve, A., J. Henrard and J.-L. Hainaut, *Co-transformations in information system reengineering*, in: *ATEM 2004*, ENTCS (2005).

[9] Cordy, J. R., *Txl - a language for programming language tools and applications.*

[10] ECSS-E-70-32, *Space engineering. ground systems and operations procedure definition language* (2004).

[11] Heering, J., P. R. H. Hendriks, P. Klint and J. Rekers, *The syntax definition formalism SDF - reference manual*, SIGPLAN Notices **24** (1989), pp. 43–75.

[12] Klint, P., R. Lammel and C. Verhoef, *Toward an engineering discipline for grammarware*, ACM Trans. Softw. Eng. Methodol. **14** (2005), pp. 331–380.

[13] Kort, J., R. Lammel and C. Verhoef, *The grammar deployment kit*, Electronic Notes in Theoretical Computer Science **65** (2002).

[14] Lämmel, R. and C. Verhoef, *Cracking the 500-Language Problem*, IEEE Software (2001), pp. 78–88.

[15] P. M. Lewis, I. and R. E. Stearns, *Syntax-directed transduction*, J. ACM **15** (1968), pp. 465–488.

[16] Petrone, L., *Syntax-directed mappings of context-free languages*, in: *Proc. Ninth Annual Symposium on Switching and Automata Theory*, 1968.

[17] Schurr, A., *Specification of graph translators with triple graph grammars* (1994).

[18] Systems, I., *Stol programmer's reference manual* (2000).

[19] Terekhov, A. A., *Automating language conversion: a case study*, in: *IEEE International Conference on Software Maintenance* (2001), pp. 654–658.

[20] Terekhov, A. A. and C. Verhoef, *The realities of language conversions*, IEEE Software **17** (2000), pp. 111–124.

[21] van Deursen, A., P. Klint and J. Visser, *Domain-specific languages: an annotated bibliography*, SIGPLAN Not. **35** (2000), pp. 26–36.

[22] Visser, E., "Syntax Definition for Language Prototyping," Ph.D. thesis, Amsterdam (1997).

[23] Visser, E., *Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9*, in: C. Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science **3016**, Spinger-Verlag, 2004 pp. 216–238.

[24] Wile, D. S., "Popart Manual," USC/Information Sciences Institute (1991).

[25] Wyk, E. V., O. de Moor, K. Backhouse and P. Kwiatkowski, *Forwarding in attribute grammars for modular language design*, in: *Computational Complexity*, 2002, pp. 128–142.

[26] Yellin, D. M., "Attribute grammar inversion and source-to-source translation," Springer-Verlag New York, Inc., New York, NY, USA, 1988.