Using Annotated Grammars for the Automated Generation of Program Transformers

Kim Mens — Diego Ordóñez Camacho

Département d'Ingénierie Informatique, Université catholique de Louvain
Place Sainte-Barbe 2, B-1348 Louvain-la-Neuve, Belgique

kim.mens@uclouvain.be
diego.ordonez@uclouvain.be

RÉSUMÉ.

ABSTRACT. When confronted with a family of different domain-specific programming languages, each with their own particular syntax but providing essentially the same semantic constructs, often the need arises to transform programs between any of these languages. This is for example the case for the domain of satellite operation languages, where every vendor or mission control centre uses its own proprietary language. In previous work, we proposed a generic technique to automatically generate program transformers between given source and target languages. Our transformer generator tool takes as input a specification of the grammar of both source and target language, tagged with specific annotations that specify the corresponding language constructs in both languages. In this paper we further validate that approach by generating program transformers between two industrial satellite operations languages. We observe that the approach falls short for more complex mappings, where a single construct in one language does not correspond directly to a single construct in the other language. To address that problem, we propose using a dedicated pre- and post-processing library and language, in which a language engineer can define how to handle such more complex mappings.

MOTS-CLÉS :

KEYWORDS: Tool generation, automatic program transformation, language engineering, annotated language grammars, operations language.

1re soumission à *IDM2007*, le 13 novembre 2006.

1. Introduction

In the domain of spacecraft mission planning and execution [QUI 04], spacecrafts need to receive specific commands from so-called operators. These commands are described in specialised programming languages called *operations languages* [CHA 06]. Operations languages 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 to be executed by a spacecraft. A large variety of operations languages exists and nearly every existing mission control centre uses its own preferred operations language to design their mission procedures. Although these operations languages can be very different from a syntactic point of view, they all provide largely the same semantic constructs and share a common foundation and programming paradigm : they are all imperative and control flow-driven languages. Figure 1 shows an example of a mission procedure expressed in the Spacecraft Test and Operations Language *Stol.*

Figure 1 – An operation in the operations language *Stol.*

```
T1 = IS_WARN("SOL_TP1")
T2 = IS_WARN("SOL_TP2")
IF (T1 || T2) THEN
    START sol_monitor()
ENDIF
```

In an attempt to facilitate the design and testing of spacecraft procedures, many operators use specialised software development tools [1]. Builders of such tools 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 use. In addition, these tools should be easily extendable to support new operations languages or newer versions of existing operations languages.

In a previous paper [Ord 06] we proposed a generic technique and proof-of-concept meta tool to semi-automatically derive translators of programs from one operations language to another, based on the grammars of those languages annotated with extra information specifying the corresponding language constructs in both languages. We performed an initial validation of the proposed approach by automatically building a translator to go from programs written in a subset of the Pluto [ECS 04] operations language to an intermediate operations language which we specially designed for the occasion. The results of that small and controlled experiment were promising and did not reveal many exceptional cases, leading us to conclude that a high-level of automation could be achieved.

An important goal of the current paper is to perform a larger-scale industrial validation of the above approach, and to further extend it to be able to deal with such industrial cases. We selected a case that was relevant to our industrial partner Rhea System [Rhe] and tried to semi-automatically build a program transformer for translating commercial procedures from the Stol [Int 00] operations language to MOIS [QUI 06].

1. MOIS 5 [QUI 06], built by Rhea System [Rhe] is an example of such a tool.

Rather than restricting to a subset of these languages, our translator considers the full extent of both languages and was tested on real-life procedures. The transformations the translator had to perform were generated automatically for most language constructs present in the Stol language. For some language constructs in the source language, however, there did not exist a direct one-to-one mapping [WIJ] to a corresponding construct in the target language. These were harder to handle automatically. Nevertheless, we managed to program by hand the necessary transformation rules for those cases to be integrated in the generated translator.

Aware of the fact that the main cause of existence of those special cases was a restriction of our technique of being applicable only to cases where there exists a one-to-one mapping between language constructs, we tried to find a solution to overcome this restriction. Following Terekhov and Verhoef's advice [TER 00], the solution we opted for was to restructure the input program passed to the translator and the output program generated by it, to enable the simpler one-to-one mappings supported by our annotated grammars technique. During our validation experiment we kept track of the different kinds of restructuring transformations we had to include manually into the translator, and eventually came up with a library of primitive restructuring transformations that could be used to facilitate the definition of appropriate mappings between many different language constructs.

After having analysed the primitive restructuring transformations in our library, we observed a potential of further simplifying the definition of these transformations by offering a dedicated high-level transformation language, backed up by the library we developed, in which a tool builder could specify the necessary program restructurings to be performed. In addition to reporting on an industrial case study of building a full program translator using our technique and tool, a second important contribution of this paper was therefore to propose using this high-level language to help defining the more complex mappings that our technique is incapable of handling automatically.

The remainder of this paper is structured as follows. Section 2 presents a brief overview of the *annotated grammars technique* for building program translators. Section 3 presents the industrial case study of producing a program transformer from the *Stol* to the *MOIS* language, reports on the results of this validation experiment and highlights some discovered limitations of the proposed technique. Section 4 then discusses how we managed to overcome the limitations encountered by building a library of transformations that can be used to enable the definition of more complex mappings. Section 5 sketches our proposal to further simplify the approach by providing a dedicated high-level transformation language. We conclude the paper in Section 7, after having discussed some related work in Section 6.

2. The annotated grammars technique

Our annotated grammars technique for the automated generation of program transformers has two main goals. First of all, we want to produce program transformers that

can *automatically* translate procedures from one operations language to another (e.g. *Stol* to *MOIS*). In other words, the produced transformers should be fully automatic and not ask for any user intervention when translating a program (as opposed to some other approaches like [LEI 03]). Secondly, the program transformer itself should be generated in an *automated* way from a specification of the source and target language. Here we deliberately use the word 'automated', rather than 'automatic', since the generation of a program transformer is not necessarily fully automatic. Indeed, although our technique can handle automatically the majority of language constructs that need to be translated, there typically remain a limited amount of less trivial cases for which some manual intervention by the user of the technique is needed.

Our technique presupposes that the languages between which we want to translate are largely similar from a semantic point of view. Furthermore, the more similar the syntax of source and target language (i.e., (the language grammars), the more automatically the program transformer can be built, and the less human intervention is required when producing the program transformer. As it requires high technical skills to declare the sometimes complex mappings between language concepts, it is our goal to make the automation as complete as possible and to make the manual effort as little as possible. Before explaining how we support the manual work, however, we first explain the basic approach that was already published in an earlier paper [Ord 06].

Figure 2 – Automated generation of program transformers : schematic overview.

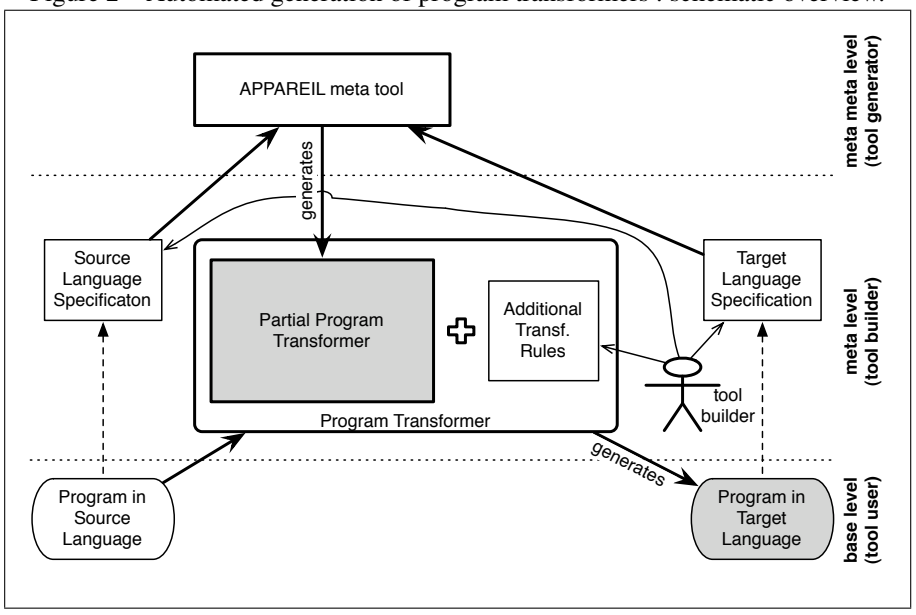


Figure 2 provides a schematic overview of our approach, indicating the three different kinds of actors involved. The bottom or base level represents the end users, who only care about having a program transformer, to which they can feed programs written in their source programming language and which automatically produces an equivalent program in the target language. At the intermediate or meta level, we have

the tool 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, they make use of our *APPAREIL* meta tool which is situated at the top (or meta meta) level. The technique used by this meta tool to automate the process of building program transformer tools is based on syntax-directed translation and transduction techniques [P.M 68, AHO 69]. The tool builders provide this meta tool with a specification of the grammar of both source and target language, tagged with annotations that specify the corresponding language constructs in both languages. Using this input, the meta tool then semi-automatically builds a dedicated transformer for translating programs from the source to the target language, and the tool builder has to intervene only to specify how to translate those cases for which no direct equivalence could be stated between productions in the source and target grammars.[2]

Now let us explain the annotated grammars technique in some more detail by working out a small example[3]. Figures 3 and 4 show the production rules for a corresponding language concept, a while loop, in the *Stol* and *MOIS* operations languages, respectively. The production rules are expressed in the SDF formalism [HEE 89], and are adorned with extra annotations. These annotations appear at two different levels. Annotations that appear at the level of the entire production are intended to specify equivalent language productions in the source and target language grammar. For example, by tagging both the production in Figure 3 and Figure 4 with the same "While" tag (inside the curly braces and 'cons()' label), we tell the *APPAREIL* meta tool that these language concepts are considered to be equivalent and that it should try to transform statements matching the production in Figure 3 to program statements matching the production in Figure 4.

Figure 3 – An annotated SDF production for the *Stol* language.

```
DO WHILE "(" cond : Expression ")" EOL
  block : Statement-List ?
  comm : CommStmt ?
ENDDO                                -> DoWhile      {cons("While")}
```

Figure 4 – An annotated SDF production for the *MOIS* language.

```
"<DecisionStep>"
   "<BooleanResult>"
     cond : MoisExpression
   "</BooleanResult>"
   "<WHILE>"
     "<REPEAT>"
       block : StepList ?
       comm : MoisComment ?
     "</REPEAT>"
   "</WHILE>"
"</DecisionStep>"                    -> WHILE      {cons("While")}
```

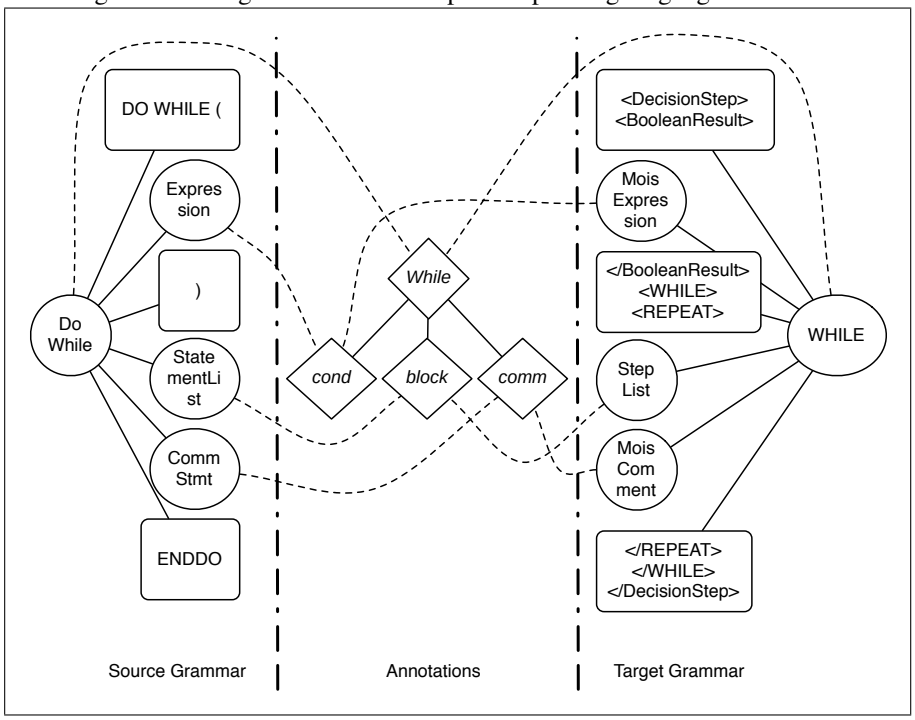
Since the left hand-side of these productions may contain several non-terminal symbols, shown in italics in Figures 3 and 4, extra annotations are needed to tell

2. A programming language is most often defined by a formal grammar, which consist of a set of production rules (or *productions*, for short) that define how strings representing valid programs in that language can be generated.
3. See [Ord 06] for a more technical elaboration of the approach.

the meta tool what non-terminals in the source production correspond to what non-terminals in the target production. (Especially when the non-terminals appear in a different order in the left-hand sides of the source and target productions.) In the example, the *cond :*, *block :* and *comm :* annotations serve this purpose. Non-terminal symbols that can be neglected are not annotated, neither are the terminal symbols.

Figure 5 illustrates how the annotations define a mapping between corresponding language constructs, used by a program transformer from source to target language. Based on the mapping defined by these annotations, our APPAREIL transformer ge-

Figure 5 – Using annotations to map corresponding language constructs.



nerator tool will produce a specific translator in *ASF+SDF* [BRA 01, KLI 93], that will act over a parsed program in the source language and produce an equivalent program in the target language. A proof-of-concept of this transformer generator was tested in a previous laboratory experiment to translate procedures written in the *Pluto* operations language, to a generic intermediate operations language designed for that experiment. That experiment, even though small and controlled, showed that a high level of automation could be achieved. Nevertheless some exceptional cases had to be treated manually by introducing additional transformation rules into the translator.

3. Industrial case study : transforming procedures from STOL to MOIS

Having performed a successful initial experiment [Ord 06], for this paper we decided to perform a more complete experiment to further validate and improve our

technique. We report on an industrial case study of building a program translator that can transform existing mission procedures written in the *Stol* operations language into the *MOIS* operations language[4]. The main difference between this experiment and our initial one is the size of the languages dealt with. We no longer restrict ourselves to a subset of an operations language, but deal with a complete source and target language. In addition, since the produced translator will be used in real life by our industrial partner, we need to test the generated program transformer on real-life procedures that cannot be freely adapted or manipulated.

Because of the larger case study we dealt with (in terms of number of productions in source and target grammar), the number of exceptional cases that could not be handled directly by our annotated grammars approach was much higher as well. On a total of 150 productions in *Stol*, which needed to be mapped to 62 productions in *MOIS*, for 109 of the *Stol* productions a 'simple' mapping to the corresponding *MOIS* production sufficed.[5] For the 41 remaining cases in *Stol*, however, such a simple mapping proved insufficient and some manual intervention by the tool builder was required to build the program transformer. Since some of those exceptional cases actually required more than one primitive transformation, as the example that follows will illustrate, in fact the total number of additional *primitive* transformations (see Section 4) that the tool builder needed to define amounted to 59.

To illustrate the kinds of manual transformations that a tool builder needs to define, let us consider an example of a more complex translation from *Stol* to *MOIS*. The language construct we are trying to transform is a variable declaration. There are two main reasons why a simple one-to-one mapping does not suffice. First of all, whereas *Stol* supports the declaration of multiple variables by means of a single declaration instruction, *MOIS* requires a separate declaration instruction per variable. Secondly, whereas in *Stol* variables can be declared anywhere in the program, in *MOIS* they must all be declared together before a procedure body, in some sort of a header. Figure 6 shows a legal example of a *Stol* program fragment that declares four variables, and its corresponding translation in the *MOIS* operations language.

Figure 7 shows the productions involved for both *Stol* and *MOIS*. Although the semantics of the corresponding language constructs is basically the same (variables are the same concept in both languages), the productions vary because variable declarations appear in different locations in the syntax tree. It is not possible to declare a one-to-one mapping between the corresponding language concepts because there is a mismatch in the shape of the abstract syntax tree (there is not an obvious isomorphic

4. In the remainder of this paper, unless explicitly stated otherwise, when we refer to *MOIS* we mean the operations language that is used internally by the MOIS tool suite, not the tool suite itself.

5. Note that different *Stol* productions are often mapped to the same production in *MOIS*. For example, *Stol* provides a different language construct for a whole range of different 'directives' (one production per directive), whereas in *MOIS* a single 'Directive' language construct exists for expressing all possible kinds of directives.

mapping, but there is a homomorphic mapping). And this example is only a simple problem : the mismatch can be more significant than for the example shown here.

Figure 6 – An example of variable declaration in *Stol* and *MOIS*.

```
--- Stol program fragment ---
LOCAL var1, var2, var3
some instruction
LOCAL var4
another instruction

--- MOIS program fragment ---
<Proc>
<Variable><name>var1</name></Variable>
<Variable><name>var2</name></Variable>
<Variable><name>var3</name></Variable>
<Variable><name>var4</name></Variable>
<ProcBody>
<instruction>some instruction</instruction>
<instruction>another instruction</instruction>
</ProcBody>
</Proc>
```

Figure 7 – *Stol* and *MOIS* productions for variable declaration.

```
--- Stol grammar production ---
(Instruction | Declaration)+                          -> StolProc
"LOCAL" Identifier DeclIdItem*                        -> Declaration
"," Identifier                                        -> DeclIdItem
[a-z0-9]+                                             -> Identifier
...                                                   -> Instruction

--- MOIS grammar production ---
"<Proc>"
Declaration*
"<ProcBody>"
Instruction+
"</ProcBody>"
"</Proc>"                                             -> MOISProc
"<Variable><name>" Identifier "</name></Variable>"    -> Declaration
[a-z0-9]+                                             -> Identifier
"<instruction>" ... "</instruction>"                  -> Instruction
```

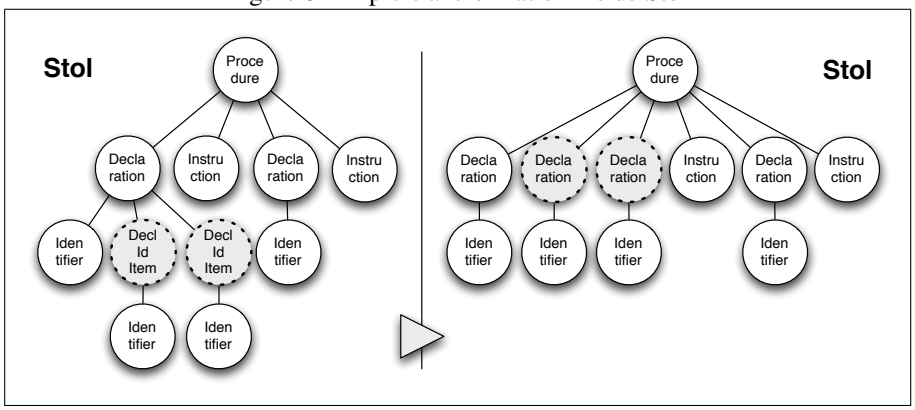
To be able to map these corresponding language constructs, the tool builder needed to perform several manual interventions when building the program translator. The first one corresponded to pre-transforming the input program to turn every multi-variable declaration into a sequence of single-variable declarations. For example, this would transform the *Stol* declaration `LOCAL var1, var2, var3` into three separate *Stol* variable declarations :

```
LOCAL var1
LOCAL var2
LOCAL var3
```

which each can be mapped without any problem to their corresponding MOIS variable declaration. The parse-tree transformation on the input *Stol* program that was executed to perform this pre-transformation is illustrated in Figure 8.

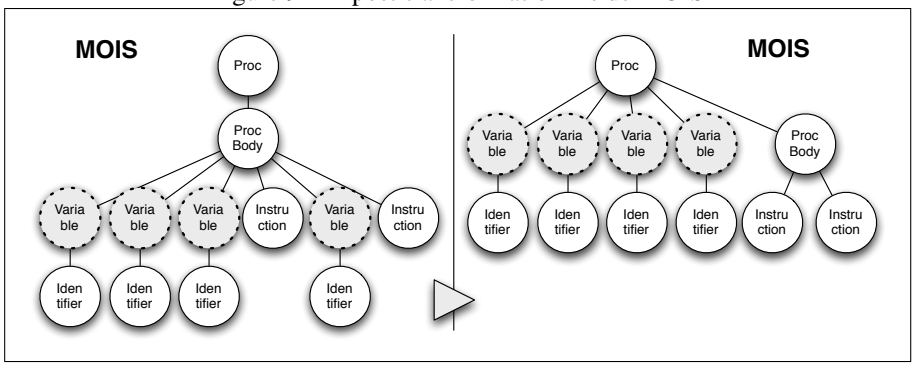
After having performed this pre-transformation, the problem still remained that variable declarations could be spread throughout a *Stol* program, whereas in the corresponding *MOIS* program they were expected to be put together in a ‘header’ at the

Figure 8 – A pre-transformation inside Stol



start of each *MOIS* procedure, before the 'body'. A second intervention required by the tool builder, therefore consisted in collecting all variable declarations, so that they could be grouped together in a single place inside the *MOIS* program. We decided to perform this transformation *after* having translated the *Stol* program to a corresponding *MOIS* program using our annotated grammar technique. More specifically, on the parse-tree of the resulting *MOIS* program, we performed the post-transformation illustrated in Figure 9, to obtain a valid *MOIS* program. (It was difficult to perform this transformation *before* the translation to *MOIS* : since *Stol* has no notion of a procedure header we could not declare where the variables need to be moved too.)

Figure 9 – A post-transformation inside MOIS



In summary, with this case study of building an industrial program transformation tool, we learned that, although our annotated grammars technique could automate an important part of building such a program transformer, quite some special cases remained where some manual intervention by the tool builder was required. This intervention typically amounted to performing some pre-transformations of the input program or post-transformations of the produced program, to enable using the one-to-one mappings supported by our annotated grammars technique. Since many of these pre- and post-transformations were similar in nature, during the case study we built a

library of typical parse-tree transformation procedures to be used by the tool builder. The next section discusses this library in more detail, before we propose how it could be turned into an actual transformation language in Section 5.

4. A library of parse-tree transformation functions

The kinds of manual interventions required by the tool builder ranged from simple replacements of incompatible characters inside text comments[6], to complex subtree modifications creating additional parse-tree nodes. As mentioned before, for dealing with the 41 'exceptional' cases in *Stol* where simple mappings between corresponding productions did not suffice, the tool builder defined 59 pre- or post-transformations of the program being translated. As summarized by Figure 10, we classified each of those 59 transformations into 5 categories of 'primitive' parse-tree transformations : *moving*, *replacing*, *mapping*, *creating* and *removing* parse tree nodes.

Figure 10 – Kinds of primitive transformations used by the tool builder.

Category	*Occurrences*
Moving nodes to different place in parse tree	7
Replacing partially/completely a node's contents	28
Mapping the type of a node to another one	8
Creating additional nodes	13
Removing nodes from the parse tree	3

For example, the transformation shown in Figure 8 was defined in terms of two primitive operations : *move* and *map*. First, we **move** up all `DeclIdItem` nodes one level, as siblings *after* their parent node ; then we **map** the type of those `DeclIdItem` nodes to the type `Declaration`. The transformation of Figure 9 was defined by collecting[7] all nodes of type `Variable` and **moving** them up one level, as siblings *before* their parent node.

For practical reasons[8], to perform the manual interventions in our case study, we transformed our parse trees into an XML DOM [W3C a], and directly manipulated them in Java. However, because DOM manipulation in Java can be pretty verbose, this made the tool builder's job of manually writing transformations quite boring and error-prone. To facilitate his job, once we realised that most complex transformations could be defined in terms of a small set of primitive operations, we decided to organise these operations into a consistent and reusable library of parse-tree transformation functions. Figure 11 shows part of the Java API for this library. By using this library instead of having to manually redefine the same low-level parse-tree manipulations

6. Keeping the comments when translating programs is a hard requirement imposed by the client. The comments tend to provide detailed information on the procedures' purpose and in this field, some mission control centers have operators that execute the procedures by hand, rather than having them executed by a machine, to have more control over the mission.
7. We do not consider the collect step as a separate operation but as an implicit part of the move operation to define the set of nodes we want to move.
8. Mainly because our industrial partner's programming team preferred this approach, because of their prior experience with XML and Java technology.

every time, the tool builder could focus on the essence of the transformations and thus became much more efficient.

Figure 11 – An API of parse-tree transformation functions.

```
void moveAfter(String nodePath, String destPath)
void moveBefore(String nodePath, String destPath)
void moveInside(String nodePath, String destPath)

            move a node described by nodePath after, before
           or inside a reference node described by destPath

Node createAfter(Node ref, String tag, Object content)
Node createBefore(Node ref, String tag, Object content)
Node createInside(Node ref, String tag, Object content)

              create a node with name tag and containing
             content, after, before or inside a node ref

Node mapTo(Node node, Document gramm, String production)

            transforms the type of a node by the type of a
          production with annotation production (as defined
            inside the annotated grammar definition gramm)

void replace(String nodePath, Object content)

              replace every node in nodePath by content

...
```

When performing his manual interventions, the tool builder needs to decide whether to perform a pre-transformation, a post-transformation, or both. Pre-transformations are performed on the input program, before it is translated by a program transformer generated by our annotated grammars technique. Post-transformations are performed on the program that is produced by that transformer. Whether to use a pre-transformation or post-transformation depends from case to case. For example, the variable declaration example of the previous section made use of a pre- *and* a post-transformation. The primitive transformation functions in our library however, are independent of whether they are used in the pre- or post-transformation phase.

To conclude this section, we show an example of what it would take to define the MOVE operation used in the post-transformation of our variable declaration example (Figure 9). Without using the library, the Java code to move the `Variable` nodes one level up would look as shown in Figure 12, whereas when using the library a simple call to the right API function suffices : see Figure 13.

5. Towards a high-level program transformation language

After having conducted our entire industrial case study and having built a reusable library of primitive parse-tree transformation functions along the way, we realized that we could take things yet a step further by offering the tool builder with a high-level dedicated program transformation language. For example, Figure 14 suggests what

Figure 12 – A verbose parse-tree transformation.

```
Node context =
   (Node) xPath.evaluate( "/Proc", document,
                          XPathConstants.NODE );
NodeList list =
   (NodeList) xPath.evaluate( "//Variable", context,
                              XPathConstants.NODESET );
for (int i = 0; i < list.getLength(); i++) {
    Node node = list.item(i);
    context.insertBefore(node, context.getFirstChild());
    }
```

Figure 13 – Transforming a parse-tree by calling the appropriate library function.

```
moveBefore("//Variable", "/Proc/ProcBody");
```

the transformations worked out in Section 3 and depicted in Figures 8 and 9 would look like in such a language. Note that we haven't constructed this language yet, but since we already have developed a full library of primitive transformation functions, developing such a language would mainly correspond to defining an appropriate parser and interpreter for the language which calls the appropriate library functions.

Figure 14 – Pre/post-transformations defined in a dedicated transformation language.

```
PRE:  MOVE "//Declaration/DeclIdItem" AFTER ".." MAPTO "Declaration"
POST: MOVE "//Variable" BEFORE "/Proc/ProcBody"
```

To be able to express this example we need the following language constructs :

MOVE path1 (AFTER | BEFORE | INSIDE) path2 moves the collection of nodes described by *path1* before, after or inside the node at the location described by *path2*. Both *path1* and *path2* are XPath expressions [W3C b] in the XML DOM representation of the parse tree over which we work. Note that *path2* is always calculated relative to the current *path1* node being moved. The *path1* node will be moved immediately **after** or **before** the *path2* node, as a sibling of it, or *inside* the *path2* node as its last child. The order in which the nodes appear in the source program is preserved when moving nodes with this operation.

For example : `MOVE "//Variable" BEFORE "/Proc/ProcBody"` moves all nodes of type `Variable` —regardless of its position in the tree— as siblings to the left of the subnode `ProcBody` of node `Proc`, while preserving the order of all collected `Variable` nodes in the source parse tree.

path MAPTO production maps the type of each of the collection of nodes described by *path* to the type of a new *production*. In fact, *production* refers to a production via the annotation that was assigned to that production. For example, if we revisit Figures 3 or 4, the production label "While" is used to refer to those productions.

Note that the **MOVE** instructions can be extended with the **MAPTO** command like this : **MOVE path1 (AFTER | BEFORE | INSIDE) path2 MAPTO production**
which, after moving, transforms the nodes collected by *path1* to the type indicated by *production*.

Other language constructs that would be needed, in our dedicated transformation language, to express the entire range of program transformations encountered in our industrial case study are :

COPY path1 (AFTER | BEFORE | INSIDE) path2 is similar to the **MOVE** instruction but creates new copies the nodes collected by *path1* rather than just moving them.

As for the **MOVE** instruction, the **COPY** instruction can be extended with the **MAPTO** command to map the newly created nodes to another type.

REPLACE path WITH nodeexpression replaces each of the nodes described by *path* with the result of *nodeexpression* (which is always calculated relative to the current *path* node being replaced) and where *nodeexpression* defines a new node to be created in an XPath-like syntax. For example, `/Declaration/Identifier(''temp'')` will create the literal "temp" inside a node `Identifier` which is nested inside a node `Declaration`.

CREATE nodeexpression (AFTER | BEFORE | INSIDE) path creates a new node, described by *nodeexpression*, before, after or inside each of the nodes described by *path*. As before, *nodeexpression* is calculated relative to the current *path* node being visited.

REMOVE path removes all nodes described by *path* from the parse-tree.

We are currently in the process of implementing this high-level dedicated program transformation language. To validate the language, we intend to define all pre- and post-transformations that were defined manually by the tool builder in our industrial case study, in terms of the primitive constructs offered by this language. This will allow us to verify if the proposed language is sufficiently expressive, easy to use, leads to simpler transformations, and so on.

Also, the language proposed above is still quite operational in nature : the tool builder uses it to describe *how* parse trees need to be transformed. As future work we will study whether we can come up with a more declarative kind of language, where the tool builder would only specify *what* parse tree expressions need to be mapped. For example : when declaring that a declaration of a sequence of variables needs to be mapped to a sequence of variable declarations, the language interpreter itself could infer the appropriate parse-tree transformations and mappings to do so.

6. Related work

Program translation, and the many problems it raises, has been the subject of several studies like those of Verhoef, Lämmel and Terekhov [TER 00, LÄM 01, TER 01b, TER 01a], many of which were used as inspiration for our approach. The core of our work is based on defining appropriate mappings between language grammars. It borrows ideas and applies principles from Lewis's syntax-directed transduction [P.M 68],

Klint's grammarware [KLI 05] and Yellin's grammar inversion techniques [YEL 88], among others.

As main medium to implement our techniques we chose the ASF+SDF meta-environment [BRA 01, KLI 93]. Nevertheless, many other alternative approaches for implementing program transformations exist, as discussed by Visser et al. in their surveys [VIS 01, WIJ]. In particular we could have chosen XSLT as an alternative implementation medium, especially when we are translating to a language like *MOIS* which has an XML-like syntax. However, the point of this paper is not about what underlying implementation technique to use, but about how to raise the abstraction level to make it easier to build automated program translators.

Some related techniques exist that address problems similar to ours. Wyk studies an expansion mechanism for modularly adding new features to a language, using attribute grammars [WYK 02]. Shurr studies graph translators, where relationships are described through additional correspondence rules [SCH 94]. Wile provides an alternative way to generate translators based on syntax-directed rules sets [WIL 91]. Moreau's framework TOM [MOR 03] is an example of how to extend a programming language like Java or C, with the necessary support to generate tree implementations and to perform tree pattern matching, in order to facilitate the manipulation of parse trees. Finally, a closely related approach, dealing with non one-to-one mappings, are Cleenewerck's *linglets* [CLE 05]. Similar to our approach, the linglets technique is highly grammar-driven and divides each translation problem in its constituent components that are each specified by means of a high-level description, and executed in separate logic steps.

7. Conclusion

In previous work we presented our annotated grammars technique to generate, from an annotated grammar specification of a source and target programming langage, an automatic transformer to translate programs from source to target language. This technique provides only a partial automation of the generation process, as it is restricted to translating between language concepts for which there exists a one-to-one mapping of the corresponding grammar productions. Nevertheless, the technique allows us to achieve a high-level of automation. However, when conducting a larger industrial case study, to build a program translator between two satellite operations languages, we observed that for translating about one third of the productions, some manual intervention was still required (i.e., a restructuring of the programs involved) in order to be able to handle them with our technique. Hence, the amount of work to be performed manually by the tool builder remains significant. In addition, when conducting our case study we experienced that this manual work was often somewhat repetitive and could therefore benefit from further automation. More specifically, we observed that the kinds of program restructurings performed by the tool builder could be defined largely in terms of a small set of primitive parse-tree transformations. Therefore, during the case study, we built a high-level reusable library of such functions,

to facilitate the tool builder's job of writing the appropriate mapping transformations. Given the positive experience gained with using this library to build a full industrial program translator, we conjecture that the tool builder's job would benefit even more from having a high-level dedicated program transformation language, backed up by our library, in which he could express the appropriate program transformations. We made an initial proposal for such a language but further research is needed to study the validity and usefulness of using the proposed language for that purpose.

8. Acknowledgements

This research, conducted in collaboration with Rhea System, was funded by the Wallonian Region in Belgium and the European Social Fund, in the context of the FIRST Europe Objectif 3 research project entitled "Une approche paramétrique de réingénierie logicielle – APPAREIL". We thank Andy Kellens, Keith Turner, Jurgen Vinju and Paul Klint for proofreading and commenting on an earlier draft of this paper.

9. Bibliographie

[AHO 69] AHO A. V., ULLMAN J. D., « Translations on a context free grammar », *STOC '69 : Proceedings of the first annual ACM symposium on Theory of computing*, New York, NY, USA, 1969, ACM Press, p. 93–112.

[BRA 01] BRAND M., DEURSEN A., HEERING J., JONG H., JONGE M., KUIPERS T., KLINT P., MOONEN L., OLIVIER P., SCHEERDER J., VINJU J., VISSER E., VISSER J., « The ASF+SDF Meta-Environment : a Component-Based Language Development Environment », WILHELM R., Ed., *Compiler Construction (CC '01)*, vol. 2027 de *Lecture Notes in Computer Science*, Springer-Verlag, 2001, p. 365–370.

[CHA 06] CHAUDHRI G., CATER J., KIZZORT B., « A Model for a Spacecraft Operations Language », *SpaceOps*, , 2006.

[CLE 05] CLEENEWERCK T., D'HONDT T., « Disentangling the implementation of local-to-global transformations in a rewrite rule transformation system », *SAC '05 : Proceedings of the 2005 ACM symposium on Applied computing*, New York, NY, USA, 2005, ACM Press, p. 1398–1403.

[ECS 04] ECSS-E-70-32, « Space engineering. Ground systems and operations procedure definition language », 2004.

[HEE 89] HEERING J., HENDRIKS P. R. H., KLINT P., REKERS J., « The Syntax Definition Formalism SDF - Reference Manual », *SIGPLAN Notices*, vol. 24, n° 11, 1989, p. 43–75, ACM Press.

[Int 00] INTEGRAL SYSTEMS, « EPOCH T&C Directive and STOL Function Reference Manual », 2000.

[KLI 93] KLINT P., « A Meta-Environment for Generating Programming Environments », *ACM Transactions on Software Engineering and Methodology*, vol. 2, n° 2, 1993, p. 176–201, ACM Press.

[KLI 05] KLINT P., LAMMEL R., VERHOEF C., « Toward an engineering discipline for grammarware », *ACM Trans. Softw. Eng. Methodol.*, vol. 14, n° 3, 2005, p. 331–380, ACM

Press.

[LÄM 01] LÄMMEL R., VERHOEF C., « Cracking the 500-Language Problem », *IEEE Software*, , 2001, p. 78–88.

[LEI 03] LEINONEN P., « Automating XML document structure transformations », *DocEng '03 : Proceedings of the 2003 ACM symposium on Document engineering*, New York, NY, USA, 2003, ACM Press, p. 26–28.

[MOR 03] MOREAU P.-E., RINGEISSEN C., VITTEK M., « A Pattern Matching Compiler for Multiple Target Languages », *12th Conference on Compiler Construction*, vol. 2622, LNCS, 2003, p. 61–76.

[Ord 06] ORDONEZ CAMACHO D., MENS K., VAN DEN BRAND M., VINJU J., « Automated Derivation of Translators From Annotated Grammars », *Electronic Notes in Theoretical Computer Science*, vol. 164, Issue 2, 2006, p. 121-137.

[P.M 68] P. M. LEWIS I., STEARNS R. E., « Syntax-Directed Transduction », *J. ACM*, vol. 15, n° 3, 1968, p. 465–488, ACM Press.

[QUI 04] QUIGLEY D., MONHAM A., « Mission Operations Preparation Management : An Effective End-To-End Approach », *IEEE Aerospace Conference*, , 2004.

[QUI 06] QUIGLEY D., CATER S. J., « Satellite Test and Operation Procedures Cost Reduction Through Standardization », *IEEE Aerospace Conference*, , 2006.

[Rhe] RHEA SYSTEM, « http ://www.rheagroup.com ».

[SCH 94] SCHURR A., « Specification of graph translators with triple graph grammars », 1994.

[TER 00] TEREKHOV A. A., VERHOEF C., « The Realities of Language Conversions », *IEEE Software*, vol. 17, n° 6, 2000, p. 111-124.

[TER 01a] TEREKHOV A., « Automating Language Conversion : A Case Study », *ICSM '01 : Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, Washington, DC, USA, 2001, IEEE Computer Society, p. 654-658.

[TER 01b] TEREKHOV A. A., « Automating Language Conversion : a Case Study », *IEEE International Conference on Software Maintenance*, IEEE Computer Society Press, November 2001, p. 654-658.

[VIS 01] VISSER E., « A Survey of Strategies in Program Transformation Systems », *Electronic Notes in Theoretical Computer Science*, vol. 57, 2001.

[W3C a] W3C RECOMMENDATION, « Document Object Model (DOM) Level 3 Core Specification Version 1.0 ».

[W3C b] W3C RECOMMENDATION, « XML Path Language (XPath) Version 1.0 ».

[WIJ] VAN WIJNGAARDEN J., VISSER E., « Program Transformation Mechanics : A classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems ».

[WIL 91] WILE D. S., « Popart Manual », USC/Information Sciences Institute, 1991.

[WYK 02] WYK E. V., DE MOOR O., BACKHOUSE K., KWIATKOWSKI P., « Forwarding in Attribute Grammars for Modular Language Design », *Computational Complexity*, 2002, p. 128-142.

[YEL 88] YELLIN D. M., *Attribute grammar inversion and source-to-source translation*, Springer-Verlag New York, Inc., New York, NY, USA, 1988.