Science of Computer Programming 75 (2010) 3–20

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Automated generation of program translation and verification tools using annotated grammars

Diego Ordóñez Camacho [a,*], Kim Mens [a], Mark van den Brand [c], Jurgen Vinju [b]

[a] *Computing Science Engineering Department, Louvain School of Engineering, 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*

ARTICLE INFO

Article history:
Received 29 April 2008
Received in revised form 23 July 2009
Accepted 7 October 2009
Available online 30 October 2009

Keywords:
Annotated grammars
Operations languages
Language families
Program translation
Automated translation
ASF+SDF

ABSTRACT

Automatically generating program translators from source and target language specifications is a non-trivial problem. In this paper we focus on the problem of automating the process of building translators between operations languages, a family of DSLs used to program satellite operations procedures. We exploit their similarities to semi-automatically build transformation tools between these DSLs. The input to our method is a collection of annotated context-free grammars. To simplify the overall translation process even more, we also propose an intermediate representation common to all operations languages. Finally, we discuss how to enrich our annotated grammars model with more advanced semantic annotations to provide a verification system for the translation process. We validate our approach by semi-automatically deriving translators between some real world operations languages, using the prototype tool which we implemented for that purpose.
© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Automatically translating source code from one programming language to another while preserving the semantics is hard [31]. Only a few of such 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 principle. In this paper we describe our approach to generate a set of automatic translators between different domain-specific languages which all operate in the same domain. Since these languages 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.

There exist several families of domain-specific languages with a common semantic basis. The operations languages used in spacecraft mission planning [21] are one such family. Database query and design languages are another example. Although we conjecture that our technique could apply to many of such families, in this particular paper we validate and illustrate our approach only on the case of operations languages.

Operations languages (OLs) are used in the domain of spacecraft mission planning. OL "programs" precisely describe procedures used to control and monitor a spacecraft. There exist quite a number of such languages, amongst which: the Spacecraft Test and Operations Language *STOL* [11,29], the Procedure Language for Users in Test and Operations *PLUTO* [8,9], the language underlying the Manufacturing and Operations Information System *MOIS* [25], the User Control Language *UCL* [2], the European Spacecraft Control Language *ELISA*, or the Test and Operation Procedure Environment *TOPE* [19]. In addition, for some of these languages more than one version exists and is currently in use. This large variety confronts the designers of spacecraft procedures with unnecessary complexity.

* Corresponding author.
E-mail addresses: diego.ordonez@uclouvain.be (D. Ordóñez Camacho), kim.mens@uclouvain.be (K. Mens), m.g.j.v.d.brand@tue.nl (M.v.d. Brand), jurgen.vinju@cwi.nl (J. Vinju).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.10.003

In an attempt to make the design and testing of spacecraft procedures easier, many operators use specialised software applications, such as the MOIS Manufacturing and Operations Information System [25]. These systems solve part of the problem by allowing operators to design and edit procedures in many OLs. However, these systems lack a generic feature that allows us to safely translate the existing procedures written in one OL to the syntax of another [5]. This obviously impedes efficient cooperation between operators or mission control centres.

Although different OLs have different syntax, overall they share many features and have a common semantical foundation. This is because they all address the same goal of communicating with a spacecraft and they all respect known standards on satellite construction and operation. Spacecraft procedures describe a set of instructions that need to be executed by a spacecraft, and therefore must have knowledge about that spacecraft and its behaviour. They need to know how to retrieve information from the spacecraft (telemetry) and how to make the spacecraft execute certain commands (telecommand). This knowledge is typically contained in a database called the Mission Information Base (MIB) [32]. The semantics of OLs is therefore limited to describing the overall flow of control of procedures, organising the way they execute the telecommands and read telemetry data.

In this paper we describe an approach, relying on ideas of grammarware development [12], to automate the generation of program translators for the family of operations languages. The approach reconciles three related problems. One is the generic language translation problem [14,30,31]. A second problem is that, when defining translators between many different languages in the same family, a large amount of translators are needed. Thirdly, there should be a way to assess the correctness of the generated translators, by verifying equivalence of the translated programs.

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 limit the excessive number of translators needed when considering a large family of similar languages, we design an intermediate representation providing a generic syntactic and semantic model common to our family of OLs. As such, we can translate from any of these languages to this representation, as well as from that representation to any other OL.

To tackle the third 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 test the control-flow equivalence of original and translated programs, by comparing their respective control-flow graphs. We generate these graphs automatically using the aforementioned lightweight annotations and then use a bisimulation relation [28] to show their equivalence.

In summary, the contributions of this paper are:

(1) A generic technique to automate the process of building translators between different operations languages;
(2) A prototype implementation of a translator generator tool based on this technique;
(3) An illustration of a common intermediate representation for a group of relevant operations languages;
(4) An extension of the technique with more semantic annotations that allows for a lightweight mechanism to verify the resulting translation.

The remainder of this paper is structured as follows. In Section 2 we set the context of our research, explaining the domain of spacecraft operations and OLs in more detail. Section 3 we briefly sketch the different steps of our proposed solution. Section 3 presents our annotated grammars technique for generating language translators and illustrates the automated generation of a translator by our prototype tool. Section 4 explains our intermediate representation language and how it improves the translation technique. Section 5 then explores how the use of more advanced annotations, from which the control-flow graph of programs can be derived automatically, can be exploited to verify the translation. We also give some directions for future research. We present our conclusions in Section 6.

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 [9]. In general, OLs 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 [9,19].

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) [32]. 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.

```
1  Initiate and confirm step Switch on Gyros
2      declare
3          event evtTimeout
4      end declare
5
6      Log "PROCEDURE Pluto_Test_43_03 Step_1";
7      Wait until ( Value of DHT30100 = ACTIVE)
8          timeout 1 h 20 min
9              raise event evtTimeout;
10
11     if (monitoring status of DHT30101 != nominal) then
12      log "Enabling Gyros command: PHC10117 scheduled 18h 43";
13         initiate and confirm PHC10117
14            with Timetag := 2008-08-02T18:43:12.000Z end with
15            refer by cmdPHC10117;
16      log "result is:" + confirmation status of cmdPHC10117;
17     else
18         log "Gyros already enabled.";
19     end if;
20
21     watchdog
22         initiate and confirm step Gyro Controller Timeout
23             preconditions
24                 wait for event evtTimeout
25             end preconditions
26
27             log "Gyro controller was not active within time.";
28         end step;
29     end watchdog
30 End step;
```

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

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.

Fig. 1 shows part of a test procedure written in the PLUTO [9] 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).

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 [16], which specifies the input–output relation of the translation and deduces the actual translator from that relation.

Our approach, initially presented in [22] 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 [23] between corresponding language constructs. The mechanism automatically generates the translator for most of the syntactic constructs, and provides support to manually extend that translator for the others.

Our technique assumes that the languages among 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 is required when producing the complete program transformer.

3.1. Preliminary experiment

During a preliminary experiment, we *manually* built a number of translators for PLUTO [9], UCL [2] and an intermediate language called IRL using ASF+SDF [33]. IRL will be explained in more detail in Section 4. We started with a subset of constructs for these languages, consisting mainly of control-flow structures, and 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 repeatable 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.

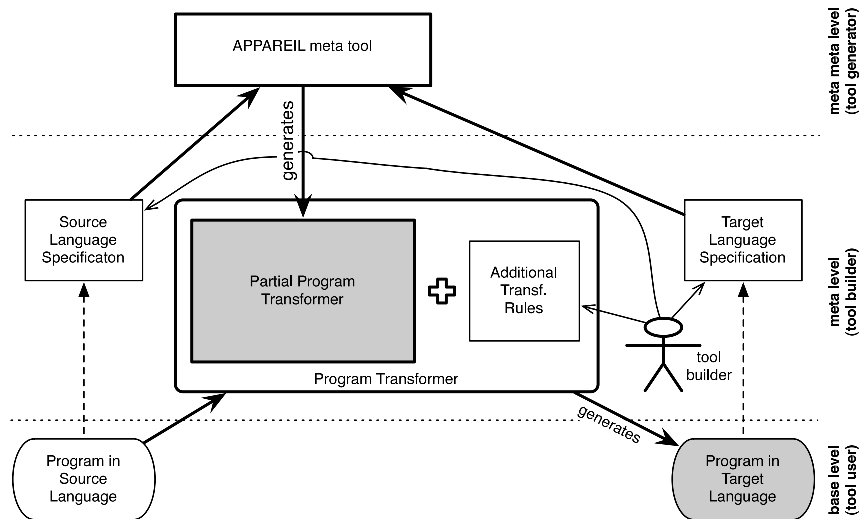

Fig. 2. Automated generation of program transformers: schematic overview [22].

In this initial experiment, we observed a high-level of repetition. In addition, declaring the sometimes complex mappings between language concepts required high technical skills. Therefore, we decided to automate the translation process as much as possible by developing an approach that could generate automatically a significant part of the rewriting rules.

3.2. The APPAREIL approach

Fig. 2 provides a schematic overview of this automated approach, indicating the three different kinds of actors involved. The bottom or base level represents the end users. They only care about having a translator which receives programs in a source programming language and which 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[1] which is situated at the top (or meta) level. 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 correspondence between language constructs in both languages. Using this input, the meta-tool then semi-automatically generates 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.

Although many existing tools could be used to implement this solution, such as DMS [3], TXL [6] or Stratego [36], we chose the ASF+SDF Meta-Environment for implementing our solution. 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 [33] 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. In the next section we discuss SDF in more detail, followed by a brief summary of ASF in Section 3.4.

3.3. SDF

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 [35].

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:

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

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

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

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

Fig. 3. An example of a production in SDF.

```
1  context-free syntax   %% (SDF syntax rules)
2
3    %% Language 1
4    "if" Expr "then" Stats "else" Stats "end if" -> If   {cons("IF")}
5    %% Language 2
6    "eval(" Cond ")" "[" Block "," Block "]"     -> Eval {cons("IF")}
7
8  context-free syntax   %% (rewrite functions)
9
10   f(If)                                        -> Eval
11   f(Expr)                                      -> Cond
12   f(Stat)                                      -> Block
13
14 variables
15
16   "$Expr$"                                     -> Expr
17   "$Stats1$"                                   -> Stats
18   "$Stats2$"                                   -> Stats
19
20 equations
21
22   []
23   f(if $Expr$ then $Stats1$ else $Stats2$ end if)%% From Language 1
24   =
25   eval( f($Expr$) ) [ f($Stats1$), f($Stats2$) ] %% To   Language 2
```

Fig. 4. An example of a simple translation function expressed in ASF+SDF.

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

$$\mathbf{label_a}: A\ \mathbf{label_b}: B\ \mathbf{label_c}: C \to X\{\ldots, cons(ConstructorName), \ldots\}$$

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. Fig. 3 shows an example of an annotated SDF production.

3.4. ASF

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 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 the variables to the matched subterms in the concrete syntax tree. See [33] for a more detailed description.

Fig. 4 first shows the SDF syntax rules for two conditional constructs in two different languages: the `If` rule[2] on line 4 defines the syntax of an if-then-else statement in the first language, whereas the `Eval` rule on line 6 defines the syntax of a conditional evaluation in the second language. Lines 10 to 12 declare some ASF rewrite functions `f` for mapping language 1 constructs to constructs in language 2. The variables (delimited by $ signs) used in the rewrite equation are declared on lines 16 to 18. Finally, the (unconditional) rewrite equation itself is defined on lines 23 to 25.

3.5. Grammar annotations

As explained above, 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 Fig. 4 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. Therefore, we declare a correspondence between them by annotating both productions with the same constructor name *cons("IF")*.

[2] This is the example of Fig. 3 from which we omitted, for brevity, the extra annotations.

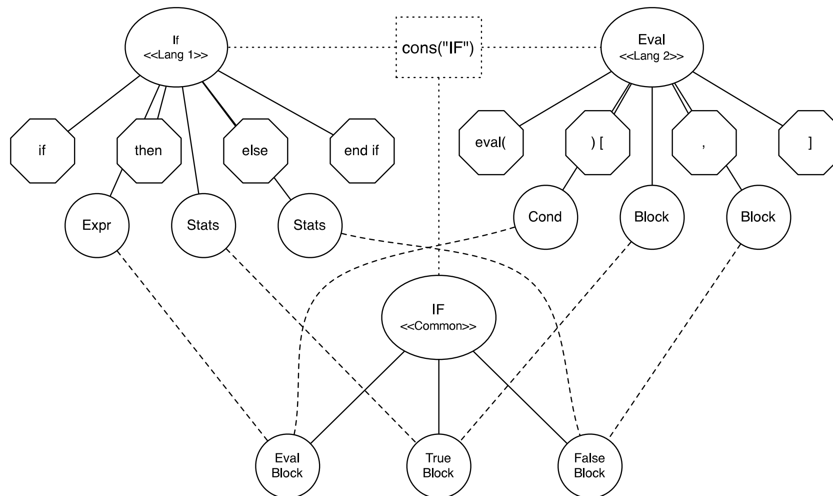

Fig. 5. Common abstract syntax tree.

As Fig. 5 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.

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 Fig. 3).

3.6. Automated generation of program translators

In Section 3.2 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 behaviour.
(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 3.7 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 3.8. The warnings generated by the meta-tool are useful here.

3.7. Transformation example

We now illustrate the approach by deriving a translator for the languages shown in Figs. 6 and 7. 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

```
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")}
```

Fig. 6. 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")}
```

Fig. 7. Part of the target grammar.

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

Fig. 8. Signatures 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. 9. Rewrite equations for translation functions.

target grammar becomes the result of that translation function. For example, for 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:

```
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 recursively.

The signatures of the translation functions (Fig. 8) and the corresponding rewrite equations (Fig. 9) are derived automatically from the grammars in Figs. 6 and 7.

3.8. *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 Fig. 6 has no equivalent production in the target grammar of Fig. 7. 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 Fig. 10 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                                   )
```

Fig. 10. Translation example.

3.9. Limitations of the approach

The simple running example we have used so far to explain the details of our approach was complemented with a more thorough validation we performed on an industrial case study. Before giving more details on this validation in Section 3.10, we first discuss some limitations of our approach.

First of all, our approach relies heavily on the existence of a straightforward mapping between the context-free grammars of source and target languages. In practice however, even when restricted to a family of highly similar languages, source and target grammars do not always align perfectly. In such situations, manual interventions like those explained in Section 3.8 are required. One example of such a misalignment which we encountered in our industrial validation, was related to the declaration of variables, when translating between the STOL and MOIS language. Whereas STOL allows us to declare variables anywhere inside a procedure, MOIS requires the use of a specific header that gathers all variables declarations. Since MOIS's notion of "declaration header" does not exist in STOL, specific transformations needed to be implemented to solve the mismatch.

Secondly, the shape of the input grammars highly influences the effectiveness of our approach. For example, suppose that one grammar was designed to be used with an LL parsing method, while the other was designed for an LR parsing method. Because both parsing methods handle priority differently, it is not always possible to map directly corresponding constructs in both languages (e.g., arithmetic expressions). Therefore, it is highly advisable first to modify the grammars to adapt them to a consistent style. This can either be done manually, or by relying on supporting tools such as the Grammar Deployment Kit [13].

Finally, we are strongly dependent on the underlying technology used to implement our approach, and we inherit all of its advantages and limitations. In particular, for implementing our prototype tool we used ASF+SDF, which has the advantage of offering predefined traversals and sort-preserving transformations. When conducting our validation, however, ASF+SDF's type-safety often proved to be a burden. Indeed, sort-preserving traversals can only emit as output exactly the same sort which they receive as input, but when translating from one language to another the sorts of the corresponding productions are often different. This being said, there will always be a trade-off between type-safety and flexibility. For example, if we would have used Stratego [36] as implementation language instead, we would have had more flexibility at the cost of less type-safety. The tool user should then be extra careful when designing his transformations.

3.10. Validation

To validate our annotated grammars approach for the automated generation of program translators, we have performed several case studies. In particular, we have conducted an industrial case study where we used our technique to build a translator transforming procedures written in the STOL operations language into procedures in the MOIS operations language.

Using the available fragments of language definitions[3] as reference (BNF portions, XML schemas, sample procedures) we first built an SDF grammar for both the STOL and the MOIS languages. The result of this process was a STOL SDF grammar composed of 150 productions, and a MOIS SDF grammar with 62 productions. The main reason for the important difference in the number of productions for both languages was the fact that STOL defines one production for every different interaction with the control centre, while in MOIS this interaction is parameterised through arguments, resulting in fewer language productions. A direct consequence of this fact was that many different STOL productions needed to be mapped to a single MOIS production.

From the 150 STOL productions, a total of 109 productions, i.e., about three quarters, could be mapped directly to a corresponding production in MOIS, by using grammar annotations. For the remaining 41 productions, some manual intervention was needed, and we had to implement 59 additional transformations, like those shown in the previous section, to complete the translator. The specific details of this particular case study were reported on in a previous paper [20].

3.11. Threats to validity

Although the industrial case study we used as validation provided sufficient positive indicators to encourage additional work in this direction, certain aspects related to the set-up of this particular case study did not allow us to make firm conclusions about the industrial applicability of the approach in other set-ups.

[3] A complete BNF was not available, as is the case for many OLs.

```
%% 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. 11. Fragment of the language definition of the Intermediate Representation Language for the family of OLs.

First, due to the strict non-disclosure and security policies typically in use in the space-operations business, the flow of information coming from the end-user of our product was very limited. Because of this, we did not receive any quantitative (e.g., the number of procedures successfully translated or not) or qualitative (e.g., possible bugs in the produced translators) feedback on the use of our tool.

Moreover, we received only ten obfuscated procedures on which to test our translators. Although these procedures were complete and complex enough to cover the majority of the constructs inside the source language, the amount of testing data remains too limited to confirm that the translator will work with any procedure.

Finally, the translated procedures were not supposed to be used for execution, but mainly for visualisation and redesigning. For this reason, some incompatibilities between source and target language were not important to consider. This was for example the case for the *goto* construct which was not present in the target language. Rather than devising a complex simulation of this construct in terms of native constructs in the target language, the end-user preferred having a straightforward mapping (i.e., treating the *goto* construct as a comment in the target language) that allowed them to visualise the construct in the translated program, without actually being able to execute it.

Because of these limitations, based on this first industrial validation we cannot emit any formal claims on the general industrial applicability of the approach yet. Nevertheless, we believe that it is a good starting point, as confirmed by the fact that the same industrial partner manifested some interest for producing another translator (for a different flavour of STOL) using our approach.

4. Intermediate representation language (IRL)

The large amount of different OLs and the need to have automatic translators between any pair of them is problematic. Even though the technique described in Section 3 offers an efficient mechanism to construct them, we still need some manual intervention. It is still a quadratic number of translators that need human attention.

To address this problem we designed an Intermediate Representation Language (IRL) that captures the commonality among a group of representative languages. We can now provide translators for each of the languages to and from that IRL. As such, we need to build only $2n$ translators instead of $n(n-1)$, where n is the number of OLs. Also for every new language only 2 extra translators are needed instead of $2n$. This is an interesting advantage, if we take into account that for the family of operations languages which we considered, there are about 20 languages for which a translator could be required.

4.1. Design of the IRL

To design our IRL we selected a representative sample of OLs like PLUTO [9], UCL [2] and STOL [11]. Due to certain language incompatibilities, abstracting the commonalities among grammars sometimes leads to a loss of information. For instance, since only one of the OLs, namely PLUTO, allows to associate a name to every "block of instructions", this information is not put in the common grammar and thus will get lost.

For example, based on the PLUTO language constructs encountered in Fig. 1 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 looks is shown in Fig. 11. The IRL uses an XML-like syntax to mark the start and the end of the different kinds of language constructs. Also note that to provide additional semantic information, we annotated the SDF representation of the IRL with extra labels and constructs.

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 sketched in Fig. 12. Language constructs common to most OLs belong to a *Core* module. Surrounding that module we have an additional layer of *Extensions*, where we add constructs that are shared by some languages but that are not general enough to merit being part of the core.

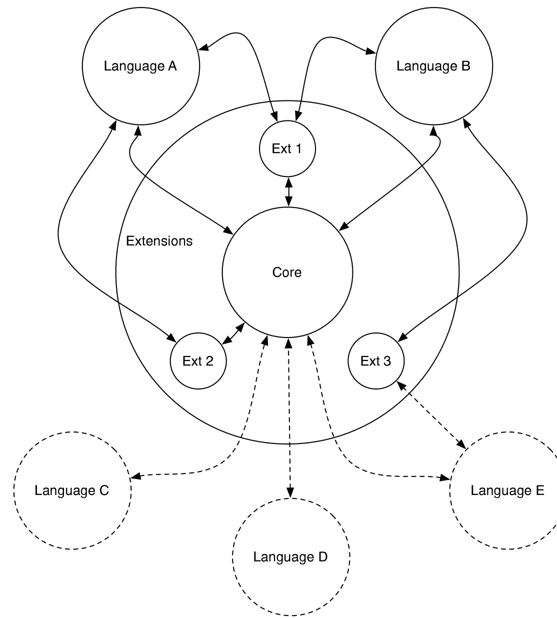

Fig. 12. Intermediate Representation Language structure.

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

Fig. 13. A "while" extension inside the IRL.

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: Ext_1 or Ext_2 to *Core*. This transformation is a rewriting rule of the form explained in Section 3.4.

As illustrated by Ext_3, there can also be cases of language constructs that are impossible to transform into *Core* language constructs, because they are too specific to a certain language or implementation (e.g., threading or exception handling). These 'unmapped' extensions will have to be managed as exceptional cases that are shared only by a subset of the languages.

4.2. Using the IRL for reusable translations

The basic idea behind the layered structure of the IRL is to obtain reuse through modularization. For every language construct present in an extension module, we provide the syntax of the productions and a semantics by mapping it to more primitive constructs in the core module.

These mappings are implemented manually. However, once an extension has been defined and a translation to the core has been specified, it can be used by the automatic translator generator directly. If any two OLs have a similar construct, a direct mapping between them can now be produced automatically, via the IRL. If a source OL has the construct, but the target has not, then a reduction via core IRL will also automatically produce a translation.

To illustrate these ideas, below we give some concrete translation examples that show the flexibility of the IRL and how to extend it with new language constructs, providing straightforward mappings while preserving generality.

Let us revisit the "generic loop statement" of Fig. 11. This is a generic construct that can express different types of loops (e.g., while-do or do-until) by using the additional terminal symbols to specify the desired semantics of a particular type of loop. For instance, by choosing the symbols `<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 map to this generic construct more difficult to understand than mappings to more concrete looping constructs. But nothing prohibits us from offering such more concrete constructs as extensions to the IRL. For example, using such an extension, a while construct could be expressed more directly and naturally as shown in Fig. 13. This extension would then transform automatically to the generic loop construct in the core IRL, producing a structure as in Fig. 14.

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

Fig. 14. A "while" instance expressed in the core IRL.

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

Fig. 15. A "do-until" extension inside the IRL.

```
%% (Step1)                         %% (Step2)
<while>                            <loop>
    A-Bool-Cond                        <checkbefore/>
    A-Stats-Block                      <loopiftrue/>
</while>                               A-Bool-Cond
                                       A-Stats-Block
                                   </loop>

%% (Step3)                         %% (Step4)
<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. 16. A (simplified) chain of transformations performing a loop inversion inside the IRL.

Now, let us consider a slightly more complex situation, where we want to translate from some language *A* that provides only "while" loops, to a language *B* that provides only "do-until" loops, by passing via the IRL. First of all, as we already explained above, we need an extension that knows how to translate "while" statements to the generic loop construct in the IRL. Secondly, we need an extension that knows how to translate from the generic loop construct in the IRL to "do-until" statements, like the one in Fig. 15. A transformation scheme to translate from a "while"-statement in language *A* to a "do-until"-statement in language *B*, via the "generic loop" statements in the IRL, is illustrated in Fig. 16. In this particular case, it requires a loop inversion.

4.3. Discussion

Using an intermediate representation or intermediate language, is a well-accepted technique for different activities, not only in the specific field of program transformation or translation, but also in other related fields. Several compilers, for instance, use *C* as an intermediate language before generating executables. If we consider the field of machine translation,[4] [10] intermediate languages have also been used as an alternative technique, as reported by Rolf [26], who promotes the idea that one of the languages, among a group of languages to be translated, can be used as the intermediate language for all translations, thus reducing the number of required translators.

In our approach, the IRL is not a full language, as in the mentioned examples, but rather an abstract representation whose structure provides enough expressiveness to deal with the languages in some family. In this sense, the IRL is closer to the Abstract Logic Language ALL [17], used to translate expressions in content languages [7], a family of domain-specific languages for multi-agent services.

The IRL is an evolving structure built from the language constructs provided by the specific family of languages we are working with. It is not a closed specification defined in advance, but is dependent on the languages we want to translate. It is able to grow and shrink as we include or remove languages from the working set. This implies that every family of languages will have its own IRL.

We are currently evolving the idea of our IRL towards a product-line approach [4]. We are enhancing its structure to include the necessary functionality to allow for better support and more reusability when defining translators between

[4] i.e., automated translation between natural languages through computer-based translation tools.

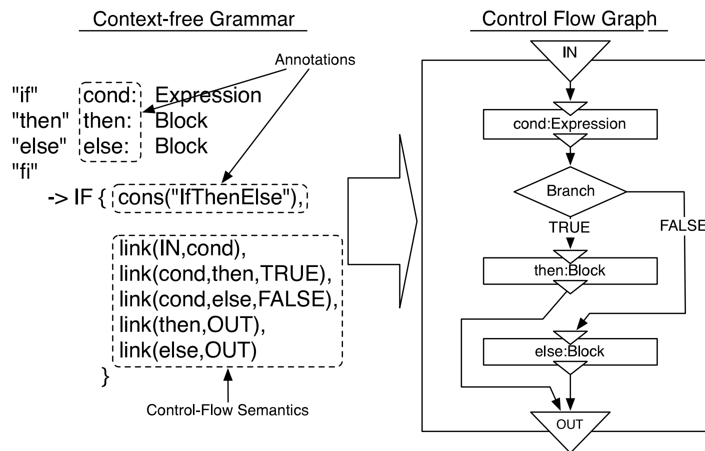

Fig. 17. The *if*-construct: annotated production and Control-Flow Graph (CFG).

different languages in the family, and to better deal with the commonalities and variabilities between those languages and translators.

5. Lightweight semantic annotations

In Section 1 we announced that our approach consisted of three main parts. The first two parts, i.e. generating an automatic translator from the annotated grammars of source and target language and providing an intermediate representation for the family of operations languages, were explained in Sections 3 and 4 respectively. This section focuses on the third part of our approach. It explains how to augment the annotations with lightweight semantic annotations. The goal is to verify functional equivalence between a source program and its translated target program.

5.1. Augmenting the annotations

The basic annotations we have described before are useful, but limited. They specify nothing more than one-to-one mappings between grammar productions for different languages. Beyond this mapping they do not convey any semantics. Even though a production may be labeled *While*, a name intentionally chosen to convey meaning to the human reader, it is just a name without any specific semantics. So, the basic annotations will not suffice as a starting point for automatic validation of transformations.

Instead, to capture a part of the intended semantics of productions we introduce lightweight semantic annotations. Although plenty of formalisms exist to precisely define the semantics of a programming language, we opted for a lightweight semantics that avoids the intrinsic complexity of these approaches. It should be kept in mind that one of our main objectives is to obtain a pragmatic approach that is easy to use by an average software engineer. More specifically, we propose to use lightweight semantic annotations that specify only the control-flow structure of the productions. This choice is motivated by the fact that, in our domain of operations languages, programs are largely control-flow driven. Procedures written in those languages describe where and when the sensitive instructions, like uploading commands and reading telemetry data, communicate with spacecrafts. These instructions must be executed in a well-defined sequence in order to successfully complete an operation. Therefore, our annotations will describe how the execution control is transferred inside a production: defining the entry point, the exit point, the branching points, and the sequence of evaluation.

Figs. 17 and 18 show on their right-hand sides a graphical representation of the control-flow graph (CFG) corresponding to the annotated context-free grammar on their left. Each grammar production is depicted by a box with an entry point IN and an exit point OUT. A box can have inner boxes representing the non-terminals inside a production. The link-annotations in the grammar define the way inner boxes are inter-connected. For sequences of productions, like for instance a list of statements, the exit point of the previous instruction links directly with the entry point of the next instruction. Assuming that every production has been annotated like this, a CFG can be generated automatically for the programs in that grammar. Further in this section we give more details on how the system builds the CFGs.

Before defining the control-flow annotations more formally, let us illustrate what they would look like by annotating the well-known imperative language constructs *If*-statement, depicted in Fig. 17, and *While*-statement, in Fig. 18.

The *if*-statement breaks the normal execution flow of instructions by offering two alternatives, of which the execution depends on the truth value of some conditional expression. In other words, after entering the *if*-production, first the *Expression* representing the condition is evaluated. Next, this result is analysed and depending on its value (true or false), either the *Block* of instructions in the then-alternative or the *Block* of instructions in the else-alternative is evaluated. Regardless of which alternative was chosen, after executing the corresponding block of instructions, the common execution path is rejoined. It is this intuition that we want to capture explicitly with the *link* annotations in the same figure.

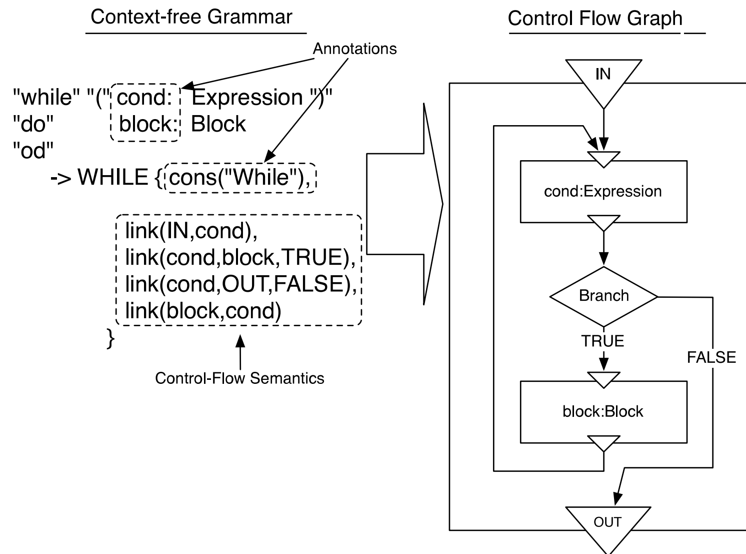

Fig. 18. The *while*-construct: annotated production and Control-Flow Graph (CFG).

Intuitively, the *link* annotations correspond to arrows in the Control Flow Graph (CFG) of the production. The first parameter of a *link* annotation refers to the source of the arrow and the second parameter to the target of the arrow. In other words, an annotation *link(source,target)* means that immediately after the expression tagged with label *source*, the expression tagged with label *target* will be executed. Two predefined labels *IN* and *OUT* are used to denote the entry and exit point of the production, respectively. For example, the annotation *link(IN,cond)* means that, when executing an expression described by this production, we start by executing the expression labeled by *cond*. Similarly, the annotation *link(then,OUT)* means that, immediately after executing the expression labeled by *then*, the execution of the expression corresponding to this production finishes. The *link* annotation can also have three arguments, in which case it represents a branch of a condition and the third argument then represents in which case that branch will be selected. For example, *link(cond,then,TRUE)* means that after execution of the expression labeled by *cond*, execution will flow to the expression labeled by *then*, in case the conditional expression evaluates to true.[5] Basically the same annotations can be reused to define the *while*-construct in Fig. 18. This figure also illustrates how loops in the control flow can be annotated.

A CFG generated by our system is a transition system composed of two types of nodes: instructions and branches. An instruction node represents a single node in the AST, that cannot be further decomposed, whereas a branch node is a point where more than one path can be chosen. Let us consider a small example of how we build the CFG for a toy program that uses a conditional *If* construct like the one in Fig. 17:

```
if (tlm1 == "OK") then {cmd1} else {cmd2} log(tlm12) fi
```

This program, after being processed with our system generates the following AST, based on some grammar of which we do not show the details here because of the simplicity of the code:

```
Block([
  If(
    Equals(
      Telemetry(tlm1),
      String(OK)
    ),
    Command(cmd1),
    Command(cmd2)
  ),
  Log(
    Telemetry(tlm12)
  )
])
```

From this AST we can generate the CFG shown in Fig. 19, which represents a transition system with two paths, one for each alternative in the branch node.

5 Note that this third argument is just an indicative tag: the value of the conditional expression will not actually be checked against that argument, since that can only be done at run-time.

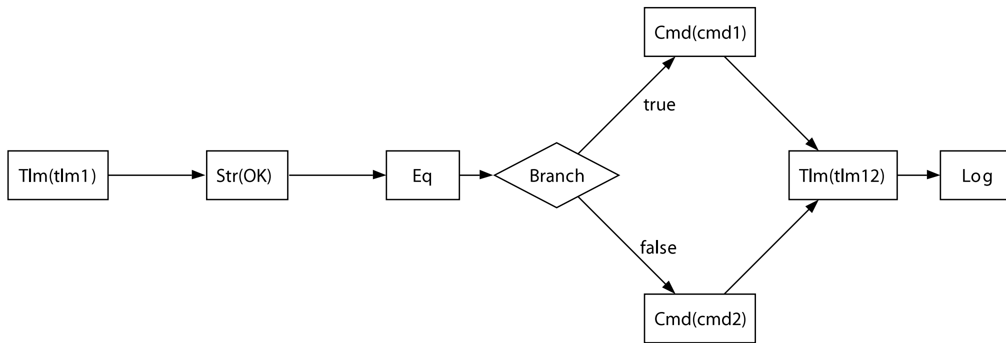

Fig. 19. CFG for a simple If construct.

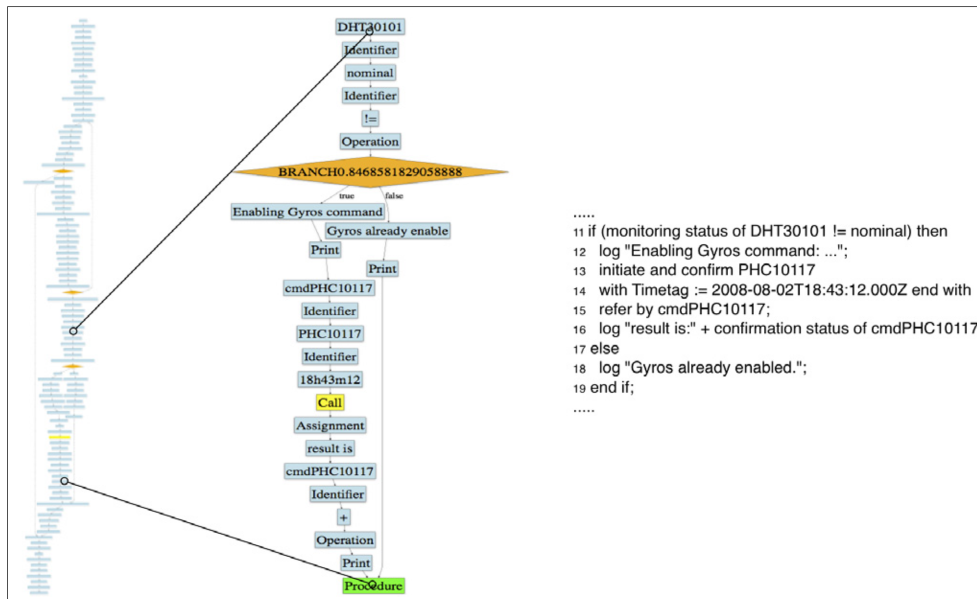

Fig. 20. CFG representation of the PLUTO code from Fig. 1.

In practice, however, concrete CFGs built from normal procedures typically consist of a large number of nodes and can have a complex structure. Fig. 20, shows what the CFG representation for the code of Fig. 1 looks like. The left of the figure shows the full CFG; in the centre we zoomed in on the portion of the graph that corresponds to the code fragment repeated on the right of the figure.

In the next section we present how can exploit these generated CFGs, to get more information on the correctness of the translation.

5.2. *Lightweight program equivalence verification*

In this section we introduce a lightweight approach to verify program equivalence, based on the CFGs presented in the previous section. We do not intend to verify full functional equivalence of programs [18], since in general the program equivalence problem is undecidable [27]. Instead, our approach is inspired on Plotkin's Structural Operational Semantics [24], and is similar to the approach followed by Veerman [34] in his work on Cobol program transformation.

Our verification technique to assess the functional equivalence of a source program and its translated counterpart, consists of the following steps:

First, we automatically generate the CFG of both source and target programs, as explained in the previous section.

Next, we reduce the CFGs by *hiding* some nodes that, from a control-flow point of view can be considered as irrelevant for the verification process. In the examples of Figs. 19 and 20, all instructions present in the program were included as nodes in the graphs. In practice, this is not a good idea because when we apply transformations to adapt the code, we most certainly will modify the structure of the programs, generally by adding, deleting or moving some instructions. Checking for equivalence under these circumstances will likely fail.

Reducing the CFGs is done by allowing the user to mark in the grammar what are the significant instructions that must be checked for control-flow equivalence. For the case of OLs these instructions are, besides the typical control-flow instructions,

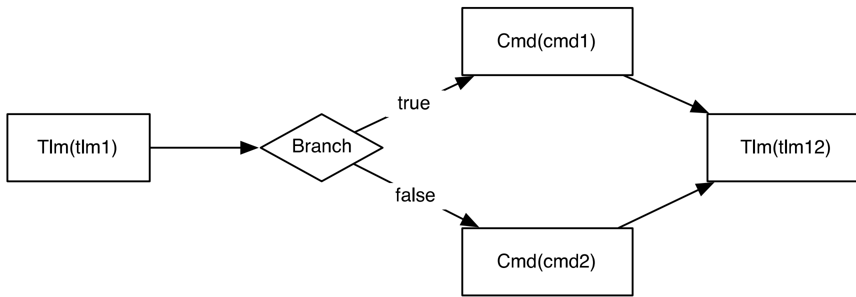

Fig. 21. CFG for a reduced If.

```
# P1                          #P2
A;                            A;
                              if(B) then {
while(B) {                      do {
  C;                                C;
  D;                                D;
}                               } while(B)
                              }
E;                            E;
```

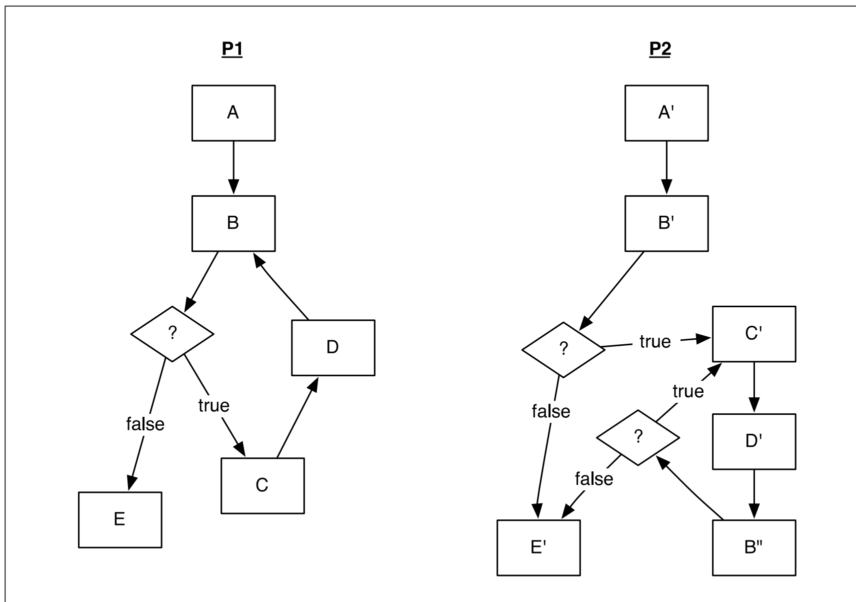

Fig. 22. Two equivalent CFGs.

telemetries, telecommands, and certain specialised functions, because they constitute the interaction with the satellite. In Fig. 21 you can see a reduced version of the CFG from Fig. 19, that only shows the telemetries, telecommands, and important control-flow instructions.

Finally, the reduced graphs are analysed to trace all their possible paths of execution. The paths extracted from the original program are compared with the paths obtained from the translated program. If for every path in the original program we can find an equivalent path in the translated program, then we say we have established a *weak bisimulation* [15] relationship between original and translated program. We can thus decide that, under these simplified conditions, both programs are *equivalent*.

Below follows an example of how two simple programs *P1* and *P2* can be verified using our technique. For convenience the instructions are simply denoted by capital letters, and are assumed to be relevant for checking flow equivalence:

The corresponding CFGs for these two programs are shown in Fig. 22, and both of them have two possible paths of execution:

```
A -> B -> E;
A -> B -> {C -> D -> B} -> E;
```

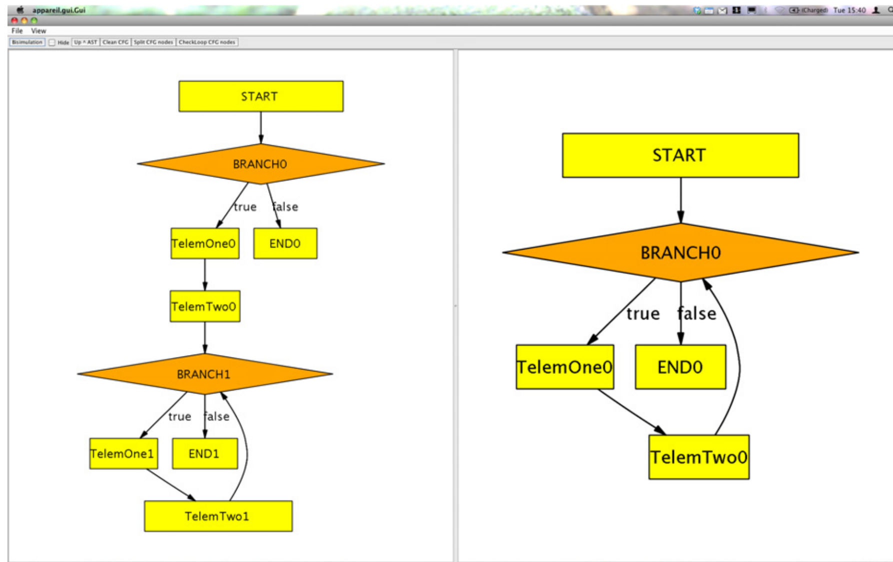

Fig. 23. A prototype of the program equivalence verification tool at work.

The fragment inside the curly braces, in the second path, corresponds to a loop that can be repeated any number of times. In our approach loops are considered as a single path, because we cannot check how many times they can be traversed. The first path, then, represents the case when the flow never gets into the loop. The second path shows the flow if the loop is traversed at least once.

The two programs, P1 and P2, are said to be equivalent under these conditions, because both provide the same possible paths of execution. We call the relation a *bisimulation* because each program can simulate the execution of the other, for every path. We also call the bisimulation relationship *weak*, because not every node in the CFGs is tested: we work on the reduced CFGs, which is equivalent to having *hidden* nodes, as in *weak bisimulation* [15].

A second example can be seen in Fig. 23, which shows a snapshot of a prototype of our program equivalence verification tool at work. The tool is checking for bisimilarity between the program on the left side of the screen, and the program on the right side. A simplified version of the program on the left is shown here:[6]

```
A;
if(B) then [C,  D] else [E, exit];
while(B) do [C, D];
E;
```

Whereas the program on the right has the following form:

```
A;
while(B) do [C, D];
E;
```

These programs cannot be considered as strictly equivalent because the left-side program has two instances of the `[C, D]` block, each with a different nature. Whereas the second instance (inside the while-do) gives rise to a looping path in the CGF, the first instance (inside the if-then) gives rise to a single pass path. Compared with the right-side program, which only has the looping path block, there is a *potential* danger that the program on the left would execute one iteration more of the `[C, D]` block than the program on the left. For cases like these, where there exists a slight possibility of inconsistency, we leave the decision in the hands of the user, and the system only emits a warning.

In general, for all cases where it is not possible to establish the proposed form of equivalence, the user gets informed about those regions in the code where potential inconsistencies have been detected. It is up to the user then, to take corrective actions to solve the detected problems. Actions to be taken for solving an inconsistency are in general related to refining the transformations included in the translator.

5.3. Limitations

Some of the limitations of our approach for assessing program equivalence are a consequence of how we extract the different possible paths from a control-flow graph. Indeed, when doing so, we abstract from the actual values of the expressions used in a program.

6 Note that the "exit" instruction, which halts the program, is not shown on the Fig. 23. It is assumed by the absence of an output link from the E node.

The abstraction results in some imprecision. Firstly, we cannot guarantee that the same number of iterations is performed in both versions of the program. Knowledge about the content of the variables controlling the loop would be required to do that. Secondly, we cannot consider any kind of time information. This could affect the control flow of constructs that have semantics dependent on time (delays, asynchronous calls, …).

Another limitation stems from the fact that we have deliberately simplified the semantics of exception handling links, to reduce the number of the possible execution paths they would generate. For example, the *try-catch* blocks, as present in the TOPE [19] operations language, is mapped to a simple two-path graph. The first path corresponds to normal control flow, while the second path corresponds to any execution that throws an exception somewhere in the block. This is a rather crude abstraction because inside such blocks, every single instruction might throw an exception generating a new path. In practice however, we have noticed that, in the space-operations domain, due to the extreme safety requirements of satellite operators, exception handling is always done on a per-instruction basis. Modern OLs like PLUTO or MOIS have not even considered it necessary to include the try-catch construct.

In summary, although our program equivalence verification technique can still be improved, the information it provides is already useful to get an additional degree of confidence on the equivalence of translated procedures. Moreover, when a lack of equivalence is suspected or confirmed, the technique can help by suggesting where to improve the mappings or the implemented transformations.

5.4. Future work

So far, we have not yet analysed every possible construct used by any existing operations language. Our semantic annotations are therefore not yet fully complete, and we still have some work ahead of us before reaching a complete enough set of annotations for OLs. If additionally we consider the possibility to use this approach for other families of languages, there will be even more possibilities still to explore and problems to overcome.

In addition to using these annotations to verify correctness of the translated programs, we are also exploring how to use the annotations in two additional directions. First, to further automate the generation of the program translators themselves. Indeed, a clever expert system may be able to propose non-trivial mappings between corresponding language productions, based on the compatibility of their annotations (i.e., based of the compatibility of the CFGs of those productions). Second, by generating a more advanced set of annotations, we can also annotate the transformations themselves with pre- and post-condition information, and produce an equivalence analysis before translating programs. This will allow us to improve the robustness and reliability of our translators.

Finally, all these extensions need to be incorporated and tested in our current prototype generation and verification tools, and the tools need to be validated on additional case studies.

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.[7]

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 the use of an intermediate representation language, capturing what is common among the group of operations languages studied, makes our translation system more extensible, scalable, modular and reusable. We have also 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.

The approach for program equivalence verification presented in this paper, 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.

[7] At the time of writing, the *PLUTO* language [9] was being put forward as a standard by the ECSS (European Cooperation for Space Standardization), whereas the language underlying the *MOIS* system [25] has been proposed as a standard to the OMG (Object Management Group).

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.

Finally, the family of operations languages was used as a case study throughout this article. We believe that our technique could be useful for other domain-specific language families as well, especially when the languages in those families have a similar semantic foundation.

Acknowledgements

This research, conducted in collaboration with Rhea System, has been supported by the Wallonian Region in Belgium, in the context of the FIRST Europe Objectif 3 research project entitled "Une approche paramétrique de réingénierie logicielle - APPAREIL" and by the Interuniversity Attraction Poles Programme of the Belgian Science Policy (Belgian State), in the context of the research project "MoVES - Modelling, Verification and Evolution of Software". We thank our colleagues at Rhea System for their help and feedback on this paper.

References

[1] A.V. Aho, J.D. Ullman, Translations on a context free grammar, in: STOC'69: Proceedings of the First Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, USA, 1969, pp. 93–112.
[2] ASTRIUM. User control language reference manual, 2003. DaimlerBenz Aerospace AG, Bremen. http://www.astrium.eads.net/.
[3] I.D. Baxter, DMS: Program transformations for practical scalable software evolution, in: IWPSE'02: Proceedings of the International Workshop on Principles of Software Evolution, ACM Press, New York, NY, USA, 2002, pp. 48–51.
[4] 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.
[5] G. Chaudhri, J. Cater, B. Kizzort, A model for a spacecraft operations language, in: SpaceOps 2006 Conference, American Institute of Aeronautics and Astronautics, 2006, p. 22.
[6] J.R. Cordy, TXL - A language for programming language tools and applications, ENTCS 110 (2004) 3–31.
[7] S. Cranefield, M. Purvis, Generating ontology-specific content languages, in : Workshop on Ontologies in Agent Systems, 5th International Conference on Autonomous Agents., 2001.
[8] B. Demeuse, S. Valera, Pluto, a procedure language for users is test and operations, Data Systems in Aerospace (1998) 307–310.
[9] European Cooperation for Space Standardisation. Test and operations procedure language, 2006. ECSS-E-70-32A.
[10] J. Hutchins, The development and use of machine translation systems and computer-based translation tools, in: International Conference on Machine Translation and Computer Language Information Processing, pp. 26–28, 1999.
[11] Integral Systems. EPOCH T&C Directives and STOL Functions Reference Manual. http://www.integ.com/, 2000.
[12] P. Klint, R. Lämmel, C. Verhoef, Toward an engineering discipline for grammarware, ACM Transactions on Software Engineering and Methodology 14 (3) (2005) 331–380.
[13] J. Kort, R. Lämmel, C. Verhoef, The grammar deployment kit, in: Electronic Notes in Theoretical Computer Science, vol. 65, Elsevier Science Publishers, 2002.
[14] R. Lämmel, C. Verhoef, Cracking the 500-Language Problem, IEEE Software (2001) 78–88.
[15] S. Lasota, Weak bisimilarity and open maps, in: SOFSEM'98: Theory and Practice of Informatics, pp. 381–388, 1998.
[16] P.M. Lewis, R.E. Stearns, Syntax-directed transduction, Journal of the ACM 15 (3) (1968) 465–488.
[17] A. Lopes, L. Botelho, Chomsky: A content language translation agent, Multi-Agent Systems and Applications IV (2005) 535–538.
[18] I.A. Mason, C.L. Talcott, Program transformation via contextual assertions, in: Logic, Language and Computation, pp. 225–254, 1994.
[19] N. Mecredy, A. Armitage, Herschel planck central checkout system system user manual, 2004. Terma Space Division. http://www.terma.com/.
[20] D. Ordóñez Camacho, K. Mens, Using annotated grammars for the automated generation of program transformers, in: Ingénierie Dirigée par les Modéles, IDM2007, Proceedings, in: Antoine Beungard, Marc Pantel (Eds.), Toulouse, France, pp. 7–24, 2007.
[21] D. Ordóñez Camacho, K. Mens, D. Quigley, J. Cater, Issues and problems in tests and operations languages translation, in: Proceedings of the SpaceOps 2008 Conference, 2008.
[22] D. Ordóñez Camacho, K. Mens, M. van den Brand, J. Vinju, Automated derivation of translators from annotated grammars, Electronic Notes in Theoretical Computer Science 164 (2) (2006) 121–137.
[23] 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, pp. 224–237, 1971.
[24] G.D. Plotkin, A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
[25] D. Quigley, S.J. Cater, Satellite test and operation procedures cost reduction through standardization, IEEE Aerospace Conference (2006) 10.
[26] P. C. Rolf, Machine translation: the languages network (versus the intermediate language), in: Proceedings of the 12th Conference on Computational Linguistics, Association for Computational Linguistics, Morristown, NJ, USA, 1988, pp. 544–548.
[27] K. Shashidhar, M. Bruynooghe, F. Catthoor, G. Janssens, Functional equivalence checking for verification of algebraic transformations on array-intensive source code, in: Design, Automation and Test in Europe. IEEE, pp. 1310–1315, 2005.
[28] C. Stirling, The joys of bisimulation, in: L. Brim, J. Gruska, J. Zlatuska (Eds.), MFCS, in: Lecture Notes in Computer Science, vol. 1450, Springer, 1998, pp. 142–151.
[29] Integral Systems. STOL programmer's reference manual. http://www.integ.com/, 2000. Integral Systems Inc. Lanham, Maryland, USA.
[30] A.A. Terekhov, Automating language conversion: a case study, in: IEEE International Conference on Software Maintenance, IEEE Computer Society Press, 2001, pp. 654–658.
[31] A. A. Terekhov, C. Verhoef, The realities of language conversions, IEEE Software 17 (6) (2000) 111–124.
[32] L.J. Timmermans, T. Zwartbol, B.A. Oving, A.A. Casteleijn, From simulations to operations: Developments in test and verification equipment for spacecraft, Data Systems in Aerospace (2001).
[33] 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, J. Visser, The ASF+SDF Meta-Environment: A component-based language development environment, in: R. Wilhelm (Ed.), Compiler Construction 2001, CC 2001, in: LNCS, vol. 2027, Springer-Verlag, 2001, pp. 365–370.
[34] N. Veerman, Towards lightweight checks for mass maintenance transformations, Science of Computer Programming 57 (2) (2005) 129–163.
[35] E. Visser, Syntax definition for language prototyping. Ph.D. Thesis, University of Amsterdam, 1997.
[36] E. Visser, Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9, in: C. Lengauer, et al. (Eds.), Domain-Specific Program Generation, in: Lecture Notes in Computer Science, vol. 3016, Spinger-Verlag, 2004, pp. 216–238.