LDTA'06

Sixth Workshop on Language Descriptions, Tools, and Applications

Vienna, Austria
April 1, 2006

Proceedings

Editors

John Boyland
Anthony Sloane

Program of LDTA'06

Session 1

- 09:00-10:00 *DSL Coordination: an Open Problem in Model Driven Engineering*
  Jean Bézivin (Invited Talk)
- 10:00-10:30 *Characterizing the Uses of a Software Modeling Tool*
  Xiaoming Li, Daryl Shannon, Jabari Walker, Safraz Kurshid, and Darko Marinov

Coffee Break: 10:30-11:00
Session 2

- 11:00-11:30 *A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars*
  José Luis Sierra and Alfredo Fernández-Valmayor
- 11:30-12:00 *AspectLISA: an Aspect-Oriented Compiler Construction System Based on Attribute Grammars*
  Damijan Rebernak, Marjan Mernik, Pedro Rangel Henriques and Maria João Varanda Pereira
- 12:00-12:30 Tool Demonstration: *The Language Evolver Lever*
  Elmar Juergens and Markus Pizka

Lunch: 12:30-14:00
Session 3

- 14:00-14:30 *Combining Deep and Shallow Embeddings*
  Joni Helin
- 14:30-15:00 *Incremental Confined Types Analysis*
  Michael Eichberg, Sven Kloppenburg, Mira Mezini and Tobias Schuh
- 15:00-15:30 Tool Demonstration: *sbp: A Scannerless Boolean Parser*
  Adam Megacz

Coffee Break: 15:30-16:00
Session 4

- 16:00-16:30 *A Domain-Specific Language for Generating Dataflow Analyzers*
  Jia Zeng, Chuck Mitchell, and Stephen A. Edwards
- 16:30-17:00 *Automated Derivation of Translators From Annotated Grammars*
  Diego Ordonez Camacho, Kim Mens, Mark van den Brand, Jurgen Vinju
- 17:00-17:30 Discussion and Closing

Foreword

The papers presented at the Sixth Workshop on Language Descriptions, Tools, and Applications (LDTA '06) are contained in this volume. These papers include both technical papers and tool demonstration papers. LDTA '06 was a satellite event of the European Joint Conferences on Theory and Practice of Software (ETAPS '06) and was held in Vienna, Austria, on April 1, 2006. Previous instances of this workshop have been held as satellite events of ETAPS in Edinburgh, UK in 2005, Barcelona, Spain in 2004, in Warsaw, Poland in 2003, in Grenoble, France in 2002, and in Genoa, Italy in 2001.

As in past instantiations, the aim of this one day workshop was to bring together researchers from academia and industry who have an interest in the field of formal language definitions and language technologies. A special emphasis is placed on the development of tools based on formal language definitions.

The program for LDTA '06 consists of 7 regular papers, 2 tool demonstrations, and an invited talk by Jean Bézivin. The regular papers and tool demonstrations were selected from 21 submissions. Each submitted paper was reviewed by at least three program committee members. The papers cover a range of topics including tools for transformation, program analysis, compilation, evolution and modeling.

We would like to thank the members of the program committee for the careful review of the submitted papers. We also thank the ETAPS organizing committee for handling the local organization of the workshop. We are again very pleased that this workshop is held in cooperation with ACM SIGPLAN and with the publication of these proceedings as a volume in the Electronic Notes in Theoretical Computer Science (ENTCS) by Elsevier. We are especially grateful to **actant** for their financial contributions for the workshop.

John Boyland
Anthony Sloane
Milwaukee (USA) and Sydney (Australia)
April, 2006

Organizing Committee

Joost Visser, University of Minho, Braga, Portugal
Eric Van Wyk, University of Minnesota, Minneapolis, USA

Program Committee

Uwe Aßmann, Dresden Technical University, Germany
John Boyland, University of Wisconsin, Milwaukee, USA (co-chair)
Jim Cordy, Queen's University, Kingston, Canada
Jan Heering, Centrum voor Wiskunde en Informatica (CWI), The Netherlands
Nigel Horspool, University of Victoria, Canada
Johan Jeuring, Utrecht University, The Netherlands
Adrian Johnstone, University of London, UK
Steven Klusener, Vrije Universiteit, The Netherlands
David Lacey, University of Warwick, United Kingdom
Brian Malloy, Clemson University, USA
Paul Roe, Queensland University of Technology, Australia
Michael Schwartzbach, BRICS, University of Aarhus, Denmark
Anthony Sloane, Macquarie University, Sydney, Australia (co-chair)
Yannis Smaragdakis, Georgia Institute of Technology, USA
David Watt, University of Glasgow, Scotland
David Wile, Teknowledge Corp, USA

Additional Reviewers

David Clarke
Wishnu Prasetya
Arjan van IJzendoorn
Jurgen Vinju
Steffen Zschaler

DSL Coordination: an Open Problem in Model Driven Engineering

Jean Bezivin [2,1]

ATLAS Group (INRIA & LINA)
University of Nantes
Nantes, France

Abstract

One of the main challenges of Model Driven Engineering (MDE) is to control the fragmentation problem resulting from the use of an important number of Domain Specific Languages (DSLs). There are several possible solutions to this problem like metalanguage frameworks, model transformation and composition operations, or other global model management techniques. This talk will consider in turn these solutions after having discussed why a general purpose unique modeling language like UML 2.0 may not adequately address all the needs in the building and maintenance of software systems. Starting from a sample of typical problems to be solved, we will question the capacity of model driven engineering to bring enhanced solutions to this field. The discussion will be based on a precise characterization of a model, broader than the classical UML-model or MOF-model definitions. A clear separation between the levels of general principles, standards and implementation platforms will be used. Some possible DSL coordination techniques will be presented to conclude the presentation.

[1] Email: `Jean.Bezivin@univ-nantes.fr`
[2] Work partially supported by the ModelWare IST Project 511731

Analyzing the Uses of a Software Modeling Tool

Xiaoming Li[1], Daryl Shannon[2], Jabari Walker[1]
Sarfraz Khurshid[2], Darko Marinov[1]

[1] *Dept. of Computer Science, University of Illinois, Urbana-Champaign, USA*
[2] *Dept. of Electrical & Computer Engineering, University of Texas, Austin, USA*
xli15@cs.uiuc.edu, dshannon@ece.utexas.edu, jlwalkr1@yahoo.com, khurshid@ece.utexas.edu, marinov@cs.uiuc.edu

Abstract

While a lot of progress has been made in improving analyses and tools that aid software development, little effort has been spent on studying how such tools are commonly used in practice. A study into a tool's usage is important not only because it can help improve the tool's usability but also because it can provide key insights for improving the tool's underlying analysis technology in a common usage scenario. This paper presents a study that explores how (beginner) users work with the Alloy Analyzer, a tool for automatic analysis of software models written in Alloy, a first-order, declarative language. Alloy has been successfully used in research and teaching for several years, but there has been no study of how users interact with the analyzer. We have modified the analyzer to log (some of) its interactions with the user. Using this modified analyzer, 11 students in two of our graduate classes formulated their Alloy models to solve a problem set (involving two problems, each with one model). Our analysis of the resulting logs (total of 68 analyzer sessions) shows several interesting observations, and based on them, we propose how to improve the analyzer, both the performance of analyses and the user interaction. Specifically, we show that: (i) users often perform consecutive analyses with slightly different models, and thus incremental analysis can speed up the interaction; (ii) users' interaction with the analyzer are sometimes predictable, and akin to continuous compilation, the analyzer can precompute the result of a future action while the user is editing the model; and (iii) (beginner) users can naturally develop semantically equivalent models that have significantly different analysis time.

1 Introduction

Alloy [6] is a first-order, declarative language suitable for expressing models of software systems, including language properties such as type systems [4]. Alloy models are amenable to fully automatic analysis, using the Alloy Analyzer [8]. The analyzer translates Alloy formulas to propositional formulas using a given *scope*, i.e., a bound on the universe of discourse, and uses off-the-shelf SAT solvers to find concrete instances or counterexamples for Alloy formulas. Alloy has been successfully used in research and teaching for several years and has assisted in finding and correcting design flaws in software systems [9]. So far, however, there has been no study into how users interact with the analyzer. It is important to study such interactions for several reasons: (1) to point out how to make the tool more usable; (2) to improve the underlying analysis technology; and (3) to develop idioms for building Alloy models that enable more efficient analyses.

Two aspects of Alloy make such an investigation particularly worthwhile: the declarative nature of the language and the bounded-exhaustive checking performed by the analyzer. Declarative logic paradigms, in general, and Alloy, in particular, tend to elicit a pervasive use of conjunction. An Alloy model is often built by first defining sets and relations that represent the model and then defining formulas that constrain the representation appropriately, starting from a minimal representation and incrementally strengthening it until a sufficient level of detail is attained. The use of the analyzer in an interactive fashion assists the users in making the incremental changes and checking their validity. These incremental changes tend to be small, so the analyzer may exploit the differences introduced between consecutive executions [1] to provide a faster analysis using the result of the previous execution.

The nature of Alloy's bounded-exhaustive checking implies that its results are valid with respect to the given scope only, i.e, if the analyzer fails to find an instance that satisfies an Alloy formula within a given scope (bound), an instance may still exist in a larger scope. For Alloy users, it is natural to increase their level of confidence in a model by increasing the scope and re-checking the model for which the analyzer previously failed to generate a desired instance in a smaller scope. Notice that in such a scenario, the only change in the model between two consecutive executions of the analyzer is the scope—once again, a situation arises where the analyzer may be able to provide a faster checking using the result of the previous execution.

This paper presents a study into how (beginner) users work with the Alloy Analyzer. We have modified the analyzer to log (some of) its interactions with the user. The modified analyzer saves a copy of the Alloy model every time the user compiles the model. To investigate the analyzer's usage, we asked students in two of our (graduate) classes to solve a problem set that required them to build Alloy models using the logging-enabled analyzer. Our analysis of the resulting logs (specifically, 68 logs from 11 students) shows three key observations:

[1] We use the term "execution" to refer to one checking of an Alloy model. The commonly used term for checking is "analysis", but in this paper, we use that term to refer to our study of Alloy executions.

(i) Users often perform consecutive executions with models that differ only slightly (which is as expected based on the two afore-mentioned aspects of Alloy);

(ii) User's interaction with the analyzer is sometimes predictable (e.g., that the user will compile and execute the model or that the user will ask for additional solutions to the model).

(iii) Users can naturally develop semantically equivalent models that have significantly different solving time.

Observation i elicits an investigation into employing incremental constraint-solving techniques to improve the analyzer's performance. Alloy's use of SAT technology and recent advances in incremental SAT solvers [13] provide a natural way to start an exploration into optimizing the Alloy Analyzer. Observation ii points out that, similar to continuous compilation and continuous testing [2,11], the analyzer can continuously precompute the result of a future action while the user is editing the model or visually inspecting a solution. The first two observations shed light on how the Alloy tool-set may be improved. Observation iii, in contrast, refutes a previous claim [12] that semantically equivalent Alloy models tend to have similar solving time: the times to solve two semantically identical but syntactically different Alloy formulas may differ substantially. While it is clear that in (almost) any reasoning system the solving time depends on the specific formulation of the problem, our results show that beginner Alloy users naturally create models that take different solving time.

The rest of the paper is organized as follows. Section 2 presents an example use of Alloy to develop and execute a model. Section 3 presents how we collected and analyzed data from the students' interactions with the analyzer. Section 4 shows some preliminary results on improving the analyzer's performance using incremental solving and continuous execution. Section 5 concludes the paper.

2 Example

We next illustrate how users interact with Alloy Analyzer to interactively develop a model. Through this example, we also introduce some key constructs of Alloy. More details of Alloy are available elsewhere [7]. As our running example, we use an interaction that a student had with the analyzer while solving a problem from the problem set. This problem considers modeling the abstract mathematical structure *tree*, i.e., a connected, acyclic, undirected graph. There are various (equivalent) definitions of a tree; we consider five definitions from a standard text-book [3]. Consider that we want to model them in Alloy to check their equivalence using the analyzer. In a typical scenario, the user of the analyzer starts from an empty model and develops it in the analyzer. To help the students, however, we provided a part of the model and asked them to provide the rest.

Let $G = (V, E)$ be an undirected graph, where V is a finite set (of vertices) and E is a binary relation on V. The following five statements are equivalent:

(i) G is a tree;

(ii) G is connected, but if any edge is removed from E, the resulting graph is disconnected;

(iii) G is connected, and $|E| = |V| - 1$;

(iv) G is acyclic, and $|E| = |V| - 1$;

(v) G is acyclic, but if any edge is added to E, the resulting graph has a cycle.

An Alloy model consists of *signature* declarations that introduce basic sets and relations, and of *formulas* that put constraints on these sets and relations. To model trees, we declare a `sig`, i.e., a set, of vertices and a binary relation on this set to represent the edges:

```
sig V {       // V is a set of vertices
  E: set V } // binary relation E: V <-> V for undirected edges
```

The cardinality operator `set` states that `E` is an arbitrary relation. (Operators `one` and `lone` respectively declare total and partial functions.) We represent an undirected edge from vertex u to vertex w as a pair of directed edges (u, w) and (w, u). Thus, `E` is a symmetric relation, which can be expressed using the transpose operator '~':

```
fact UndirectedGraph { E = ~E } // E is symmetric

fact NonEmpty { some V }        // consider non-empty graphs
```

A *fact* introduces a constraint that must be satisfied by any *instance* of the model, i.e., any satisfying assignment of values to sets and relations. The fact `NonEmpty` requires the instances to have at least one vertex[2]. The formula `some e` evaluates to true if the expression `e` evaluates to a non-empty set. (Similarly, `no e` evaluates to true when `e` evaluates to the empty set.)

We express Statement 1 using a *predicate*, i.e., a parameterized formula that can be *invoked* elsewhere:

```
pred Statement1() { Connected() && Acyclic() }

pred Connected() { all disj v1, v2: V | v1 in v2.^E }

pred Acyclic() { all v: V | not InCycle(v, E) }

pred InCycle(v: V, c: V -> V) { v in v.c ||
  some v': v.c | let c' = c - (v -> v') - (v' -> v) | v' in v.*c' }
```

[2] This condition is required for equivalence of Statements 1–5.

Alloy provides the usual logical operators: `&&` (and), `||` (or), `not`, `=>` (implication), and `<=>` (bi-implication). The keyword `disj` requires `v1` and `v2` to be different; `all` and `some` respectively represent universal and existential quantification; `in` represents subset (each expression is semantically a set and thus `in` does not represent set membership) ; '`^`' denotes transitive closure, and '`*`' denotes reflexive transitive closure. The expression `v2.^E` thus denotes the set of all vertices reachable from `v2` following one or more traversals along (edges in) `E`, and `Connected` states that there is a path between any two distinct vertices. The predicate `InCycle` states that a vertex `v` is a part of a cycle according to an edge relation `c` iff there is a self-loop at `v` or `v` has some neighbor `v'` such that even if we remove the edge connecting `v` and `v'`, these two vertices are still connected. The operators '`->`' and '`-`' represent pairing (more generally, Cartesian product) and set difference, respectively.

Our problem set asked the students to extend the above Alloy model to represent each of statements 2–5.[3] The students also had to express the equivalence of the five statements in Alloy and to check them using Alloy Analyzer. We next present the interaction that a student had with the analyzer to solve the above problem. We chose this particular interaction as it is representative for the steps that the users perform while working with Alloy, going through a cycle of modifying the model and executing it.

The user first checked that the above model is consistent using the following:

```
assert Test { !Statement1() }

check Test for 3
```

An Alloy *assertion* introduces a formula that should be checked, in this case that `Statement1` does not hold. The command `check` instructs the analyzer to find a counterexample to the given assertion using the specified scope, specifically 3. The analyzer proceeds by looking for satisfying assignments to the negation of the formula.[4] Each such assignment effectively gives a valuation to the set `V` and the relation `E` to satisfy the negation of `Test` (and implicitly all `fact` formulas).

When the analyzer finds a satisfying assignment, it can visually present it (as a user-customizable graph, where nodes represent specific objects and relations are represented with edges). The user can also choose to generate more satisfying assignments for the given formula. (This option in the analyzer exploits solution enumeration in SAT solvers such as mChaff [10] and relsat [1].)

The user further formulated Statement 3 and checked its equivalence with

[3] More precisely, the problem also asked the students to write the predicate `Connected`.

[4] Besides `check`, Alloy Analyzer also provides a command `run` that directly finds satisfying assignments for a given formula; `run Statement1 for 3` is equivalent to the above and would avoid the double negation, but the student likely forgot about the command `run`.

Statement 1:

```
pred Statement3() { Connected() && #E = #V - 1 }

assert Test { Statement1() <=> Statement3() }

check Test for 3
```

`Statement3` uses the set cardinality operator '`#`' to (incorrectly) represent the constraint $|E| = |V| - 1$. Alloy Analyzer finds a counterexample for the above formula. The issue is that our Alloy model represents each undirected edge using two directed edges. Note that the counterexample would not have been found in the scope of one, which would allow only one element in the set `V`. Users typically start checking with the scope of three or four: smaller values can miss many counterexamples, and larger values lead to large execution time.

The user quickly realized the mistake and corrected the formula:

```
pred Statement3() { Connected() && #E = #V + #V - 2 }
```

This simple step illustrates the power of the analyzer: the users can automatically analyze their models for correctness (without the given scope). Quick gaining of feedback helps users to correct their models as they build them. Indeed, it is the full automation of the analysis that encourages the users to interactively build the models in small steps and with frequent executions.

The user next used `check Test for 10` to check the model within the scope of ten. Although `Statement1` and (corrected) `Statement3` are equivalent for all graphs with up to three nodes, on the evidence so far, they may be non-equivalent for larger graphs. This increase in the bound from three to ten is somewhat unusual; users typically increase the value for one or two.

The user then added Statement 4:

```
pred EV1() { # E = #V + #V - 2 }

pred Statement3() { Connected() && EV1() }

pred Statement4() { Acyclic() && EV1() }

assert Test { Statement1() => Statement3()
              Statement3() => Statement4()
              Statement4() => Statement1() }

check Test for 4
```

Note that the user realized that the equivalence of several statements can be expressed using a circular implication.[5] (The lines without `&&` (or any other connective) are implicitly conjoined, so the three implications in `Test`

[5] That is also how the equivalence is proved in text-books.

are conjoined.) This check revealed no counterexample, so the user increased the scope from 4 to 5.

The user then proceeded by adding statements 2 and 5 and after a few more checks arrived at the following:

```
// connected but removing an edge makes it disconnected
pred Statement2() {
  Connected()
  no E or
    all v1, v2: V | (v1 -> v2) in E =>
      let E' = E - (v1 -> v2) - (v2 -> v1) |
        some disj v3, v4: V | not v3 in v4.^E' }

pred Statement5() { // acyclic but if any edge is added, cyclic
  Acyclic()
  all v1, v2: V | not (v1 -> v2) in E implies
    let E' = E + (v1 -> v2) + (v2 -> v1) |
      some v: V | InCycle(v, E') }

assert Test { Statement1() => Statement2()
              Statement2() => Statement3()
              Statement3() => Statement4()
              Statement4() => Statement5()
              Statement5() => Statement1() }
```

The final model includes also the latest definition of the above formulas.

Using the Berkmin SAT solver [5], the Alloy Analyzer checks the final assertion for all graphs with up to 4 vertices and reports no counterexamples. The SAT solver completes its search in 4.08 sec on a Pentium M 1.8GHz processor.

3 Study

This section presents the study that we performed to analyze how (beginner) users interact with the Alloy Analyzer. We first describe our experimental setup. We then present how we modified the analyzer to log its interaction. We next discuss an analysis of the resulting logs. We finally show that equivalent Alloy models can require significantly different solving times.

3.1 Experimental Setup

We collected the logs from the graduate students working on problem sets in two graduate seminars at the University of Texas at Austin and the University of Illinois at Urbana-Champaign. The students had no experience with Alloy prior to the classes but were given about two and a half lectures on Alloy in the class. The problem set consisted of two problems. One problem was our running example on modeling tree definitions and checking their equivalence.

The other problem was to model and solve a puzzle [14], given in English, to assign eight different jobs to four people subject to a list of constraints.

We told the students how to enable the analyzer to collect the logs of their model developments. We also told them that we may use the (anonymized) models that they develop as case studies in an investigation of how users work with the analyzer and how to develop *incremental* techniques to provide faster solving. We did not tell the students the specific experiments that we want to perform. Submission of logs was voluntary and did not affect the student's grade, either positively or negatively. Their solutions were graded only based on the final models that they sent.

3.2 Logging

We design our logging facility to provide the Alloy developers with usage data that may help further improve the Alloy tool-set. The current logging facility logs build, execution, and user-interface events. All the information required to replay an event is stored together with the time stamps to record when the event begins and ends. Besides the time stamps, the information stored includes the configuration of the Alloy Analyzer and the source file being built (and any source files referenced). Two types of events are used in this paper: (1) BUILD, which records how users build a model, and (2) COMMAND, which records the related information about how users executes commands after a successful compilation. In order to replay an event, we record the configurations of the Alloy Analyzer and the SAT solver and the string representation of the command.

Though only two events are used in this paper, our logging facility records other usage data that might improve further understanding of the usage pattern of the Alloy Analyzer. For example, the user interface events may help us streamline the workflow of the analyzer. The information about failed builds may shed light on the understanding the common mistakes users make while learning how to use the Alloy Analyzer and how tools can help them develop correct models.

3.3 Analysis

We next present our analysis of the logs collected in our two classes. In total, we collected logs from 11 students. (Many students either worked offline or used logging incorrectly and thus didn't provide us with useful logs.) There were a total of 68 UI sessions and 2308 compilations in these logs. Of these compilations, 391 (or 16.9%) failed with compile errors and 452 (or 19.5%) were successful compilations but without any execution. Unfortunately, our logging did not record the models of failed compilations as we had not expected them to be useful for improving the Alloy Analyzer's performance. However, their relatively large percentage suggests that the beginner users may have problems learning some constructs of Alloy. We plan to record failed models in the future; it would be interesting to analyze them to identify potential improvements in the language or its documentation to avoid common mistakes. Apparently, the users quickly learned how to deal with the compile errors: the users started compiling models more frequently to catch errors early in the recently changed parts of the models.

We next analyze the 1465 logged successful compilations and executions to detect what changes the users made to the models between the consecutive executions. We call a change an *event*. We first introduce the types of events that our analysis detects. We then describe how our analysis detects these events using a level of syntactic and semantic comparisons. (These are not full syntactic and semantic comparisons, as explained later.) We finally present the analysis results.

3.3.1 Events

Recall that Alloy models consist of signatures (which correspond to data in programs), formulas (which correspond to code in programs), and commands (which correspond to the inputs in program runs). An important part of the command is specifying the scope, i.e., the bounds for the basic sets in the model. We define the following events to track the changes in the parts of the model:

- For signatures: SN (sig_new) adding a new sig; SD (sig_delete) deleting a dig; SM (sig_mod) modifying an existing sig.
- For formulas: FN (formula_new) adding a new formula; FD (formula_delete) deleting a formula; FM (formula_mod) modifying an existing formula.
- For commands: CN (command_new) adding a new command; CD (command_delete) deleting a command; CM (command_mod) modifying an existing command.
- For scope: OS (only_scope) the only change in the model is changing the scope in the command; ND (non_decrementing) the scope was increased only; OO (one_one) only one bound was increased for exactly one.
- Summary events: SS (single_sig) only one sig was changed in the model; SF (single_formula) only one formula was changed in the model; CR (consecutive_repeat) two *consecutive* executions have the same signatures, formulas, command, and scope; ER (execution_repeat) an execution is repeated but not necessarily consecutively.

We have written a program that traverses a given log (or a set of logs) and counts the number of events. The program proceeds as follows. It first removes from the model semantically unnecessary syntactic elements such as comments and white spaces. It then parses parts of the model and uses two types of comparisons: syntactic comparison (for SN, SD, SM, FN, FD, FM, CN, CD, CM, SS, and SF events) and semantic comparison (for OS, ND, OO, CR, and ER events).

3.3.2 Syntactic Comparison

The syntactic comparison in our program uses the concept of *text similarity*. The program splits each semantic unit—signature, formula, command—into two parts, the declaration of the unit and the content of the unit. The program compares these units separately. For example, the following sig definition:

```
one sig V extends W {
  E: set V }
```

is represented as a 3-element tuple (`V`, `E: set V`, `one, extends W`), where `one` and `extends` are Alloy keywords that specify singleton sets and subsets, respectively. The similarity of two signatures is then a weighted sum of the similarities of the three components; our current implementation uses equal weights for these three components.

Our program uses the *edit distance* as the metric for similarity between two component strings. The edit distance between two strings is the number of keystrokes required to change one string to the other string. We use the edit distance as our goal is to find what the user changed between consecutive executions. The edit distance is normalized to the lengths of the two strings. When the edit distance is 0, it means the two strings are identical. When the edit distance is 1, it means the two strings are totally different. The values between 0 and 1 represent the similarity of the two strings. The lower the edit distance, the higher the similarity. For example, when the edit distance is 0.5, we need to change about 50% of the one string to get the other string.

An advantage of syntactic comparison is that it traces certain changes more closely than the semantic comparison. For example, if the user changes the definition from `sig V` to `one sig V`, the internal Alloy representations for the two versions are quite different, while syntactic comparison can easily find that the user just changed a single sig definition. However, the limitation of syntactic comparison is that it requires further parsing when difference in smaller granularity is desired, for example to detect if only scopes are changed in the commands.

We can define two semantic units to be equal, modified, or different based on the similarity between their components. Two semantic units are equal if the edit distance between them is 0. Our program uses an empirically selected threshold of 25% to determine if two units are modified versions or simply different. (We determined the threshold by a detailed manual inspection of comparisons for several randomly selected examples.) If the edit distance is below the threshold, the units are treated as modified versions. Otherwise, they are different. Our threshold is pretty high such that the confidence of counting two versions as modified is high, i.e., the two versions have only minor differences. In other words, the data shown in Figure 3.3.4 is a slight underestimation of the true number of modification cases, which means that we underestimate the potential that incremental solving can bring to Alloy Analyzer.

3.3.3 Semantic Comparison

The semantic comparison in our program parses the model and performs a level of semantic analysis to detect the changes that the user made. Specifically, our program detects, based on the scope, the bound for each basic signature. Our running example with trees had only one signature, `V`, but in general there can be several signatures in the model. The scope is then a vector of the bounds for each of these signatures. The user can specify the scope in several ways in the commands. The full details are elsewhere [7], and we provide here only a few examples: `check Eq for 5` specifies that the bounds for all signatures should be five, `check Eq for 4 V` specifies that `V` should have bound four while other signatures should have the default values (currently three), and `check Eq for 5 but 3 V` specifies that all signatures should have bound five except that `V` should be three. Our program analyzes the command and the signatures to build the entire scope vector. It then compares these scope vectors between different executions of Alloy models.

3.3.4 Number of Events

Table 3.3.4 shows the number of events that the users performed while changing the Alloy models. For each of the 19 UI sessions with most builds, we tabulate the total number of events performed. About 60% of the modifications between two consecutive executions involve only one formula, and 17% of the consecutive executions differ only in their scopes. These numbers highlight the importance of incremental solving in the Alloy Analyzer. Moreover, 11% of all model executions are identical to some previous execution in the same UI session. (While similar consecutive models are akin to spatial locality, repeated models are akin to temporal locality.) This suggests that the analyzer could keep the results of executions and compare each new model with the previously executed models.

3.4 Equivalent Models, Different Performance

We next show that semantically equivalent, but differently expressed, Alloy models can require significantly different solving times. Our result refutes the claim made by Sullivan et al. [12][page 140]:

> TestEra, because it employs the Alloy Analyzer's translation to SAT, is largely insensitive to the constraint's logical structure.

While it is clear that we could artificially construct equivalent Alloy models that have different solving times, we consider the models that the students actually submitted as solutions to the problem set. Specifically, we consider the solutions submitted for the problem on tree equivalence, used as our running example.

Recall that the problem asked the students to model five definitions of a tree and to express their equivalence. While the students came up with several different formulas to express the definitions, they also came up with five different formulas to express the equivalence. This was much to our surprise, as we expected that the students may use only two or three formulas to express equivalence. This diversity points out that the students likely did not copy the solutions. More seriously, the diversity shows how beginning users can surprise expert users (or even tool developers) by using the tools in a way that was not anticipated. Finally, the diversity provides an additional motivation to understand how users work with the Alloy Analyzer.

To present the students' approaches to checking equivalence, we use $\texttt{S}_i$ to stand for $\texttt{Statement}_i\texttt{()}$, where $1 \leq i \leq 5$. The students used four different formulas for equivalence:

```
assert Eq1 { S1 => S2 && S2 => S3 && S3 => S4 && S4 => S5 &&
             S5 => S1 }
assert Eq2 { S1 <=> S2 && S1 <=> S3 && S1 <=> S4 && S1 <=> S5 }
assert Eq3 { S1 <=> S2 && S2 <=> S3 && S3 <=> S4 && S4 <=> S5 &&
             S5 <=> S1 }
assert Eq4 { S1 <=> S2 && S1 <=> S3 && S1 <=> S4 && S1 <=> S5 &&
             S2 <=> S3 && S2 <=> S4 && S2 <=> S5 &&
             S3 <=> S4 && S3 <=> S5 &&
             S4 <=> S5 }
```

One student used a rather different approach for checking equivalence; instead of representing the equivalence of all statements in one formula, the student used four formulas:

```
//Uncomment one line at a time to check equivalence.
//assert Eq5 { S1 <=> S2 }
//assert Eq5 { S1 <=> S3 }
```

user	SN	SD	SM	FN	FD	FM	CN	CD	CM	SS	SF	OS	ND	OO	CR	ER	#C
1	0	0	0	19	7	50	20	18	23	0	47	17	12	5	3	14	92
2	0	0	0	2	2	21	8	9	14	0	23	13	9	4	4	16	50
3	0	0	0	1	1	34	14	11	3	0	25	5	4	1	1	8	48
4	0	0	0	0	0	18	7	7	5	0	18	4	4	2	1	5	35
5	0	0	0	0	0	19	10	13	11	0	19	8	6	5	1	6	43
6	0	0	1	0	0	34	0	0	0	1	34	6	6	0	6	9	41
7	0	0	0	1	0	23	2	2	20	0	18	4	4	1	0	4	36
8	0	0	0	2	0	35	8	2	12	0	19	5	4	0	2	4	39
9	7	8	3	25	17	35	5	4	16	3	38	10	6	2	1	3	60
10	0	0	0	0	0	48	6	6	10	0	42	9	5	0	0	0	57
11	0	0	0	2	1	42	6	6	18	0	36	18	12	5	1	1	63
12	0	0	0	1	0	35	9	9	20	0	28	18	13	6	2	3	57
13	0	0	0	0	1	21	9	9	6	0	20	4	2	1	1	1	34
14	0	0	0	3	2	25	4	2	3	0	28	4	2	0	1	6	34
15	0	0	0	0	0	29	1	0	17	0	25	12	10	6	0	0	43
16	8	13	0	32	22	29	14	9	12	0	32	5	5	0	3	7	52
17	0	0	0	1	0	38	0	0	3	0	26	8	7	0	2	4	37
18	2	8	2	11	2	58	5	5	15	2	57	9	6	0	2	20	76
19	19	11	8	31	15	17	7	7	4	4	27	6	6	0	0	0	42

Table 1
Number of changing events the users performed while modifying the models.

```
//assert Eq5 { S1 <=> S4 }
assert Eq5 { S1 <=> S5 }
```

We next compare the performance of the analyzer for checking the above assertions of equivalence. Note also that all those assertions are equivalent among themselves; as a matter of fact, they are all equivalent to true. Thus, the negation of the assertions is unsatisfiable, and the analyzer cannot find any solution for the negation (and a counterexample for the formula). We use each assertion with the same model for expressing tree definitions and check the assertion for the scope of four. For `Eq5`, we check all four assertions separately and sum all four times. Checking for scope 4, the Alloy Analyzer takes 12.6 seconds for `Eq1`, 10.7 seconds for `Eq2`, 16.2 seconds for `Eq3`, 28.6 seconds for `Eq4`, and 20.1 seconds for (all four) `Eq5`. Thus, an appropriate formulation gives a 2.6X speed-up in the solving time. When we check the same assertions for scope 5, the speed-up increases to 5.7X.

This result points out that the users should be made aware that different models can result in greatly different solving time. Actually, expert Alloy users gain this through experience and do rewrite their models in order to speed up the execution. We leave it as a future work to study these rewrites to generate a set of guidelines for (re)writing the models to obtain efficient executions.

4 Potential Improvements

This section shows some *preliminary* results that illustrate potential for improving Alloy Analyzer's performance. We present two types of improvements: (i) improvements that can be obtained by using *incremental SAT solving* to speed the execution of (similar) consecutive models, and (ii) improvements that can be obtained by computing some results while the user is editing the model or visually inspecting a solution.

4.1 Incremental Solving

As shown earlier, Alloy users often execute similar models one after the other. This leads us to consider the use of incremental SAT solvers to improve the analyzer's execution time. Incremental SAT solvers work as follows [13]. They first take one SAT problem, as usual, and find whether it is satisfiable or not. Additionally, they track how the inference steps that they perform depend on the input clauses. After that, the next SAT problem can be presented to the solver not from scratch, but as a *delta* from the previous SAT problem, which describes what old clauses to remove and what new clauses to add. The solver can then use this information to invalidate the inference steps that depended on the removed clauses and to perform the search only for the new clauses. Solving only the delta often results in a much improved SAT solving time, compared to the SAT solving time of the new problem from scratch.

We present results that exploit incremental SAT solving in two steps performed with the Alloy Analyzer: (i) adding only one fact to the model and (ii) increasing the scope by one. These are only preliminary results for evaluating the potential of incremental analysis. To obtain the exact results, we would need to modify the entire analyzer to perform incremental analysis.

4.1.1 Adding a Fact

Alloy users can add constraints to their model by adding new facts. A user may add a fact while building the model or while tuning the generation of instances (or counterexamples). For example, recall the following assertion from Section 2:

```
assert Test { !Statement1() }
check Test for 3
```

The analyzer's execution generates a tree with one vertex and no edges. Assume that we instead want to see larger trees, say a tree with exactly three vertices. We can add the following fact to express this requirement:

```
fact { #V = 3 }
```

Re-running the analyzer at present requires translating the modified Alloy model in its entirety into a CNF formula and then solving the formula. Using an incremental SAT solver instead, we can simply provide the solver the delta using an incremental translation that produces only the boolean formula that represents the new constraint. Doing so not only allows the solver to generate a desired tree more efficiently but also eliminates the need to translate the whole Alloy model into CNF.

In our experiments, we modified the Alloy compiler so that the compiler can translate the delta fact into the delta boolean formula. By comparing two consecutive models, we can find the new fact in the later model. Our modified compiler renames this fact such that the original Alloy compiler generates the CNF formula only for the new fact. Our modifier compiler also instructs the SAT solver to reuse the solving trace of the previous model. The translation between the previous model and the boolean variables is maintained when generating the delta boolean formula, so that the user can visualize how the new fact affects the original solution. Our modification ensures the correctness of delta solutions, i.e., we improve the performance by reusing the previous solution, but the new solution satisfies both the models with and without the delta fact.

Figure 1 shows the performance gain of using our incremental compiling. We present 10 cases from the logs where the user only adds one fact to a model between consecutive executions. These 10 cases have significant solving time for the original model and thus the speed-up can be observed; if the original model takes little solving time, we do not need incremental solving. We measure the time to solve the original model (t_{orig}), the time to solve the later model with the delta fact in its entirety ($t_{orig+delta}$), and the time to solve

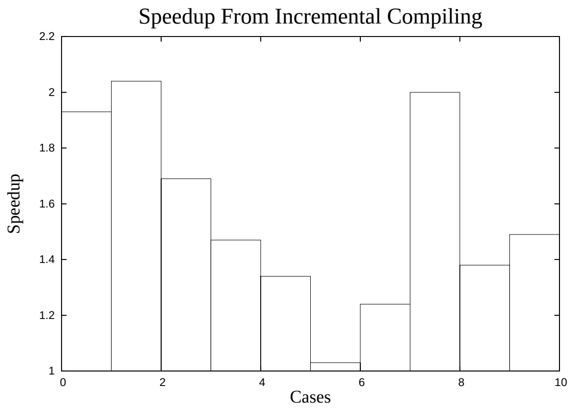


Fig. 1. Speed-up.

the later model incrementally, i.e., to solve only the delta fact (t_{delta}). The speed-up is given by $\frac{t_{orig+delta}+t_{orig}}{t_{delta}+t_{orig}}$. The speed-up of incremental compiling ranges from 1.03 to 2.04 with an average of 1.56.

4.1.2 Increasing Scope

Incremental solving need not be performed only using incremental SAT solvers. The analyzer's result from a previous execution can be used instead to re-write the current model so that it induces faster analysis even when the whole model is re-compiled and the SAT solver is executed from the beginning.

Consider checking an assertion with the analyzer. If it fails to find a counterexample, Alloy users are likely to increase their level of confidence in their model by increasing the scope and re-executing the analyzer, as illustrated by following consecutive executions from a student log:

(i) `check Test for 4`

(ii) `check Test for 5`

Since Alloy's analysis is *scope monotonic* (i.e., if the analyzer fails to find a counterexample using scope i, no counterexample exists for scope $j \leq i$), when the user executes the check for scope i after executing the same check for scope $i-1$, we can use the fact that no counterexample was found during the first execution to direct the analyzer to check for *exactly* i vertices. To illustrate, the time to check the equivalence formulation in Section 2 for scope 5 reduces from 8 min 43 sec to 8 min 24 sec. Although in this case the improvement is only 3.5%, we believe that a better technique could yield higher speed-ups.

Our logs show that about 13% of all commands only increase the scopes compared with the previous execution. Of those, 31.7% commands increase the scope only by one on exactly one signature. The data indicates that users frequently increase scopes, usually with small step. Thus, incremental solving for increasing scope might improve the performance of the Alloy Analyzer.

4.2 Continuous Execution

Continuous compilation [2] is a method for reducing the latency time in Integrated Development Environments: while the programmer is editing the program, the machine is compiling it in the background, and thus when the programmer wants to execute the program, it is already compiled. A similar technique has been recently proposed for testing. Continuous testing [11] runs the unit tests in the background as the programmer is editing the program; if a test fails, the programmer is warned that his recent change may be breaking some regression tests.

We propose to use a similar approach in Alloy Analyzer: it can continuously execute the analysis whose results the user will (likely) ask for next. One situation in which this naturally applies is while the user is editing a model. The analyzer can then be translating the model into SAT and running it on the underlying SAT solver. After asking for a solution, the user would then be presented with it faster. Another, somewhat surprising, situation where continuous solving applies is while the user is visually inspecting one solution for a model: the analyzer can then be instructing the SAT solver to generate the next solution in the background. Our results show that (beginner) users are not very likely to check for the next solution, but if they do check the next solution, they tend to repeat this operation a few times in a row. Repeatedly looking at the next solution is something that we have anecdotally observed in expert Alloy users as well.

We next present results that estimate the decrease in the latency for getting next solution with and without continuous execution. We cannot obtain the precise result, because our logs do not record the entire user's interaction with Alloy Analyzer and thus we do not know the precise time when the users performed all actions. Our logs record when a user begins checking for the next solution and when the SAT solver returns the result. We use $begin_i$ to denote the beginning time of the ith checking and end_i for the time when the results are returned to the user. Thus, $begin_{i+1} - end_i$ is the period when the user inspects the ith solution, which is also the potential decrease in the latency for getting the $i + 1$th solution if we can overlap the computation for the $i + 1$th solution and the user inspection.

We examined 84 cases in which users check for the next solution. On average, users spend 8 seconds examining the returned solution and wait less than 1 second for the SAT solver to return the next solution. In all 84 cases, the time users spend on visual inspection is longer than the time the SAT solver generates the next solution. If the analyzer instructs the SAT solver to search for the next solution immediately after the previous solution is returned, the user can get the next solution instantaneously when the next solution is desired.

5 Conclusions and Future Work

We have presented an analysis of the use of Alloy Analyzer, a tool for automatic analysis of software models written in Alloy, a first-order, declarative language. Although there has been a lot of prior work on Alloy, there has been no study of how users interact with the analyzer. We analyzed the interactions that 11 graduate students had with the tool while developing two models for a problem set. Our results show that: (i) users often perform consecutive executions with slightly different models, and thus incremental analysis could speed up the interaction; (ii) users' interaction with the analyzer are sometimes predictable, and the analyzer can precompute the result of a future action while the user is editing the model; and (iii) (beginner) users can naturally develop semantically equivalent models that have significantly different analysis time.

Our results provide an encouraging starting point for the further analysis of Alloy Analyzer. We are planning to collect more logs and analyze them to detect potential further improvements for the analyzer. We are also planning to implement full incremental analysis and continuous execution in the analyzer. Finally, Alloy Analyzer is only one example tool used in software development. We believe that studies of tool usage are important, and we are planning to explore usage of other tools in the future.

Acknowledgments

We would like to thank Derek Rayside and Greg Dennis for their help with Alloy logging.

References

[1] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real world SAT instances. In *Proc. National Conference on Artificial Intelligence*, pages 203–208, 1997.

[2] Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *Proc. 17th International Symposium on Parallel and Distributed Processing (IPDPS)*, Washington, DC, 2003.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[4] Jonathan Edwards, Daniel Jackson, Emina Torlak, and Vincent Yeung. Subtypes for constraint decomposition. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, July 2004.

[5] Evgueni Goldberg and Yakov Novikov. BerkMin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, March 2002.

[6] Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. http://sdg.lcs.mit.edu/alloy/book.pdf.

[7] Daniel Jackson. *Alloy 3.0 Reference Manual*, May 2004. http://alloy.mit.edu/reference-manual.pdf.

[8] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[9] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.

[11] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[12] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–142, 2004.

[13] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: A new incremental satisfiability engine. In *Proc. 38thConference on Design Automation (DAC)*, Las Vegas, NV, June 2001.

[14] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning (2nd ed.): Introduction and Applications*. McGraw-Hill, Inc., New York, NY, USA, 1992.

A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars

José Luis Sierra[1] Alfredo Fernández-Valmayor[2,3]

Dpto. Sistemas Informáticos y Programación
Fac. Informática. Universidad Complutense
Madrid, Spain

Abstract

In this paper, we describe PAG (*Prototyping with Attribute Grammars*), a framework for building Prolog prototypes from specifications based on attribute grammars, which we have developed for supporting rapid prototyping activities in an introductory course on language processors. This framework works for general non-circular attribute grammars with arbitrary underlying context-free grammars, includes a specification language embedded in Prolog that strongly resembles the attribute grammar notations explained in the course cited, and lets students produce comprehensible prototypes from their specifications in a straightforward way.

Key words: attribute grammars, language prototyping framework, education in language processors, Prolog

1 Introduction

Attribute grammars have been recognized as very valuable artifacts to bring together the design of programming languages and the construction of their processors [13][15][20]. We have also adopted attribute grammar methodology as the basis for our pedagogical strategy in teaching a graduate level introductory course on language processors at the *Complutense* University of Madrid (Spain). In this course, we encourage a clear distinction between the specification of the source language and its translation, and the subsequent implementation of the processor. During specification, students are compelled to use attribute grammars, and subsequently to apply systematic techniques

[1] Email:`jlsierra@sip.ucm.es`

[2] Email:`alfredo@sip.ucm.es`

[3] The Spanish Committee of Science and Technology (TIC2002-04067-C03-02, TIN2004-08367-C02-02 and TIN2005-08788-C04-01) has partially supported this work.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

to move to a one-pass top-down or bottom-up based implementation. We also promote an intermediate prototyping stage, where students program a prototype in Prolog that closely mirrors the specification. This stage is founded both on a technical basis (e.g. to let the student validate and improve the quality of the specifications) and on a pedagogical one (e.g. to motivate the student to undertake an otherwise unpleasant activity). Our choice of Prolog instead of more specific environments [14][20] is because our students study logic programming as a core undergraduate topic, and therefore they are familiar with the use of this language for developing small- and mid-scale programming projects.

We have been promoting Prolog's definite clause grammars (DCGs) [1] as a prototyping technique for several years. While DCGs have been a very useful mechanism in letting students comprehend the main concepts behind syntax-directed translation techniques, and even many fundamental concepts behind logic programming, we have also detected several practical limitations: the lack of support for left-recursive specifications, and the need to be aware of the evaluation order for semantic equations. To overcome these limitations we have developed PAG (*Prototyping with Attribute Grammars*), a Prolog framework for the rapid prototyping of language processors from their attribute grammar based specifications. PAG includes a specification language that closely resembles the usual notation for attribute grammars that we use in our lessons. The combination of a general parsing algorithm with a simple technique for attribute evaluation lets PAG accept general non-circular attribute grammars with arbitrary (even ambiguous) underlying context-free grammars, a must for a successful prototyping activity.

The structure of this paper is as follows. In section 2 we describe the pedagogical context where PAG has arisen. In section 3 we overview PAG. In section 4 we describe its implementation. Finally, section 5 provides some conclusions and some lines of future work.

2 The Pedagogical Context

In this section we describe the pedagogical context of the present work, centered on our course on *Language Processors* at the *Complutense* University. In subsection 2.1 we briefly outline the main aspects of this course. In subsection 2.2 we describe how we have addressed rapid prototyping using Prolog's DCGs, as well as their pedagogical strengths and weaknesses. Finally, in subsection 2.3 we establish a set of initial requirements for a prototyping framework that preserves the advantages and overcomes the limitations of the DCG-based approach, and we justify the design and construction of PAG on the basis of these requirements.

2.1 The Course on Language Processors

Language Processors is a two-semester course of the Graduate Degree in Computer Science at the *Complutense* University. Our main pedagogical goal is to let students learn methods for the systematic description of computer languages and for the systematic development of their processors. Therefore, in this course we teach how systematically to *specify* computer languages and their processors, and how to systematically *implement* these processors using different standard techniques: hand-coded and automatically generated predictive-recursive top-down translators, automatically generated table-driven top-down translators, and also automatically generated LR bottom-up translators [2][8].

In our pedagogical method, we adopt a problem-based learning approach where students, working in groups, incrementally solve the problems posed by the specification and the construction of an interpretative compiler of a Pascal-like language. In order to facilitate the maintenance and evolution of the language and of its processor, we promote a clear distinction between specification and implementation. As said before, the central descriptive formalism used during specification is based on attribute grammars, although we also use other complementary resources (e.g. regular expressions and/or finite automata for describing lexical aspects, and semiformal algorithmic specifications for describing target machines and their supported object languages).

During the teaching of the course we have noted how students are reluctant to assimilate the convenience of separating specification and implementation. To convince them of the advantages of this separation of concerns we have adopted the following strategies:

- We make the incremental development process model usually followed in the construction of language processors explicit. Indeed, we start by proposing the implementation of a processor for a minimal language (two primitive types, an expression language involving basic operators with different precedence and association rules, declaration of variables and the assignment statement). Once students have constructed this processor, we propose successive extensions of the basic language: control statements, user-defined types, and recursive subprograms.
- We require several alternative implementations. The processor for the minimal language is initially hand-coded as a predictive-recursive descendent translator. Once we have introduced the students to more advanced implementation techniques, they must refactor the translator in terms of the techniques introduced , using suitable domain-specific supporting tools. In addition, they incorporate the successive extensions to the language proposed, either in the hand-coded processor or by using one of the tools tested.
- We introduce a rapid prototyping activity.

While incremental development and alternative implementations are useful in order to appreciate how carefully prepared specifications can pay off during the development process, we have realized that they are not enough to convince students of their benefits. Indeed, it is not unusual to see how excellent students, overwhelmed by the work to be done, concentrate on the programming tasks while abandoning specification activities. This situation has been largely alleviated by rapid prototyping. Indeed, this activity is highly motivating for students, since they are able to get a running processor at a very early stage of the development process. Therefore, students feel that they are doing worthwhile work during specification, and they concentrate on this activity. This effort has very positive repercussions in the rest of the process. The next subsection concentrates on rapid prototyping in our pedagogical method.

2.2 Rapid Prototyping with Definite Clause Grammars

Prolog DCGs have been largely recognized as valuable artifacts for prototyping language translators [5][24][25]. We have also realized this fact as part of our teaching experience. Indeed, as said before, we have adopted DCGs as a basic prototyping technique for several years, since our students have a good working experience with Prolog as part of their undergraduate education. This enables us to introduce the technique as syntactic sugar for the direct Prolog encoding of a translation schema in one or two one-hour classroom sessions.

As we have realized during the use of the technique, our students have found the use of DCGs very valuable for understanding the main concepts behind language translation. More concrete than attribute grammars, DCGs have also helped many of them to better understand the operational mechanisms behind the more abstract attribute grammar-based specifications. In addition, we have been pleasantly surprised to discover how DCGs have helped some of our students to better understand some of the more important features of logic programming: non deterministic execution and the use of unification to deal with incomplete structures [24].

Regardless of these advantages, the approach also exhibits several limitations, as exposed in the introduction. Prolog's DCGs do not work with left-recursive underlying context-free grammars (with the exception of specialized implementations that make use of *tabling*, like [26]), therefore hindering many otherwise natural specifications (e.g. left-recursive syntax for expressions with left-associative operators). Also the usual DCG style promotes attribute evaluation during parsing, which forces students to be aware of the evaluation order for semantic equations. Thus, the primary spirit of an attribute grammar-based specification is broken.

Example 2.1 In Fig. 1 we illustrate the use of DCGs in the construction of prototypes based on attribute grammars. Notice that, in order to transform the specification into a suitable form for prototyping, the student must make an effort comparable to transforming the attribute grammar into a syntax-directed translation schema oriented to a top-down recursive-descent imple-

(a)

```
exp ::= exp + term
   exp0.v = exp1.v + term.v
exp ::= term
   exp.v = term.v
term ::= num
    term.v = num.v
term ::= ( exp )
    term.v = exp.v
```

(b)

```
exp(Vo) --> term(V1),rexp(V1,Vo).
rexp(Vho,Vo) --> [+],term(V1),{Vh1 is Vho+V1}, rexp(Vh1,Vo).
rexp(V,V)  --> [].
term(V) --> [num(V)].
term(V) --> ['('],exp(V),[')'].
```

Fig. 1. (a) A very simple attribute grammar; (b) DCG-based prototype resulting from (a). Left-recursion has been eliminated and an explicit evaluation order for the semantic equations has been chosen.

mentation.

The limitations exposed can be frustrating for the average student, thus defeating his/her acceptance of general attribute grammars as a good way to think about programming language design and implementation. Indeed, we have realized that many students concentrate on producing specifications that avoid left-recursion in the underlying grammars, which can be readily translated onto DCG-based prototypes, but which in some cases are rather unnatural and not suitable for producing certain types of implementations (e.g. based on LR translators).

2.3 A better Prototyping Alternative

The limitations detected with the use of DCGs during prototyping have led us to consider alternative approaches. Among the initial requirements for a suitable alternative we established the following:

- *Simplicity* requirement. The selected approach should maintain the simplicity of DCGs. It should be easily assimilated by our students and the impact on the current course schedule should be minimized. Ideally, the approach should provide a very simple formalism, close to the notation used for attribute grammars in our lessons (see Fig. 1a for an example of such a notation).
- *Syntactic freedom* requirement. The approach should be able to deal with arbitrary context-free syntax.
- *Semantic freedom* requirement. The approach should deal with general non

circular attribute grammars. Indeed, in our introductory course we do not deal with the possibility of circular attribute grammars, and we identify circularity as an erroneous condition.

- *Comprehensibility* requirement. The generated prototypes should be easily understood by students, who should be able to trace their behavior when required.
- *Deployment* requirement. The supporting tool should be portable and easy to install. In addition, it should be modular and easy to integrate into web-based learning scenarios like [22], and those based on the learning object paradigm [21][23], since we are making intensive use of e-learning solutions in order to accommodate a smooth migration of our pedagogical methods to the forthcoming European Space of Higher Education [7].

When looking for a suitable solution meeting all these requirements, we considered the following alternatives:

- Using programming languages that, like Elegant [11] or ALE [4], are derived from or closely related to the attribute grammar formalism. Nevertheless, this alternative clearly violates the *simplicity* requirement, since we must spend a considerable amount of time teaching the new language to our students.
- Using an existing attribute-grammar based environment, like FNC-2 [12], Eli [9] or Cocktail [10]. Some of them, such as LISA [17] are recognized as especially well-suited for educational purposes [18]. Nevertheless, this kind of environments is usually conceived as development tools instead as prototyping ones. They usually integrate parser generators for deterministic classes of context-free grammars (e.g. LL(1), LALR(1), etc.), which violates the *syntactic freedom* requirement. Also they are oriented to generating efficient static attribute evaluators, which violates the *semantic freedom* requirement, and, more important, the *comprehensibility* one (although comprehensibility can be enhanced by using appropriate GUI support, as in LISA [18]).Finally, these systems integrate specifications languages with powerful features (e.g. attribution patterns, multiple inheritance, template rules, etc). While these features are very valuable during development, they violate the *simplicity* requirement in our educational context.

In addition, all these third-party alternatives also could make deployment in a web-based learning scenario more difficult than our own tool. Once we concluded this exploration without finding the *perfect* candidate, we decided to undertake the design and construction of PAG. The rest of the paper describes the technical details of the resulting framework.

3 The Prototyping Framework

PAG produces working prototypes from attribute grammar specifications and suitable Prolog implementations of the semantic functions required. In this section, we survey the framework. In subsection 3.1 we describe the structure of specifications in PAG. In subsection 3.2 we describe how the framework is used during prototyping.

3.1 Specifications in PAG

A specification in PAG is formed by two parts:

- The specification of the attribute grammar. This specification is given in a Prolog-embedded domain-specific language that strongly resembles the basic notation for attribute grammars used in a typical introductory course on language processor construction.
- The definition of the semantic functions. This definition can be kept independent of the attribute grammar, and relates the signatures of the semantic functions with the Prolog goals used to compute them.

Specification ::= *Symbols* *Axiom* (*Rule*)+
Symbols ::= (nt‘(’**symbol**,**inh-attr-list**,**syn-attr-list**‘)’. |
 t‘(’**symbol**,**attribute-list**‘)’.)+
Axiom ::= axion‘(’**symbol**‘)’.
Rule ::= **head-nt** ‘::=’ *Body* (, *Equations*)?.
Body ::= [ ] | **symbol** (, **symbol**)*
Equations ::= *Equation* (,*Equation*)*
Equation ::= *Attribute* = **definition**
Attribute ::= **att-name** of **symbol**

Fig. 2. Syntax of the specification language.

The syntax for the PAG attribute grammar specification language is outlined in Fig. 2. This syntax, which is embedded in Prolog with the usual facilities to introduce user-defined operators, is featured as follows:

- Non-terminals must be declared using the `nt/3` predicate. The first argument is the symbol itself. The second argument represents the inherited attributes, while the third declares the synthesized attributes.
- Terminals can be declared using the `t/2` predicate. The first argument is the terminal name, while the second one is a list with the lexical attributes. Notice that a terminal without lexical attributes does not need to be declared.
- The axiom of the grammar is distinguished using the `axiom/1` predicate.

- Attributes attached to syntactic symbols are referred to using the `of` operator. When there is more than one occurrence of a symbol in a production, the occurrence number can be indicated (by default it is the first one).
- Grammar rules are built with the `::=` operator. λ is specified with the empty list `[]`, as in DCGs. With these rules it is also possible to attach a set of semantic equations, which are specified using the `=` operator.
- The left-hand side of a semantic equation must be an attribute reference. The right side of a semantic equation can be an arbitrary Prolog term, which will usually contain references to other attributes. This term will be interpreted as the expression for computing the attribute value.

Example 3.1 In Fig. 3 we show an attribute grammar for a simple calculator language based on DESK, the example language introduced in [20]. This language also enables us to bind constants to values and to use these constants in the binding scopes. The attribute grammar associates a suitable value to each expression.

In addition to the specification of the attribute grammar, definitions need to be provided for the semantic functions used in computing the attribute values. PAG establishes the `defun/2` hook for this purpose. In this predicate, the first argument must be a term whose functor identifies the function name, and its arguments are associated with the function inputs. The second argument of `defun/2` is associated with the function result. In its definition, the body of the corresponding clause will link the function with a Prolog computation of the result. By default all the functions declared are *strict* (i.e. their arguments in semantic equations will be evaluated before applying the function). This default behavior can be altered by distinguishing the function signature with the `nonstrict/1` hook. In this case, the evaluation strategy must be customized in the definition. Finally, any undeclared semantic function will be interpreted as declared as `defun(F,F)`, and therefore as a term constructor. PAG defines several utility functions in a prelude file, which can be loaded with each specification.

Example 3.2 In Fig. 4 we include the definitions of the semantic functions used in the grammar of Fig. 3. The `+` arithmetic function is already defined in the prelude, and we only include it for the purpose of illustration. The `emptyEnv`, `mkEnv` and `valueOf` functions are used to manage a simple environment binding variables to their values.

3.2 Prototyping with PAG

PAG lets students automatically process the specifications introduced in previous subsections to generate prototypes. PAG is able to deal with general non-circular attribute grammars with an arbitrary underlying context-free syntax in a simple way. The structure of the prototypes generated is sketched in Fig. 5, and it is featured as follows:

```
nt(prog, [], [val]).
nt(exp, [envh], [val]).
nt(fact, [envh], [val]).
nt(constPart, [], [env]).
nt(constDefs, [], [env]).
nt(constDef, [envh], [env]).
t(num, [val]).
t(id,  [lex]).
axiom(prog).

prog ::= exp, constPart,
  val of prog = val of exp,
  envh of exp = env of constPart.
exp ::= exp, +, fact,
  val of exp(1) = val of exp(2) + val of fact,
  envh of exp(2) = envh of exp(1),
  envh of fact = envh of exp(1).
exp ::= fact,
  val of exp = val of fact,
  envh of fact = envh of exp.
fact ::= id,
  val of fact = valueOf(lex of id, envh of fact).
fact ::= num,
  val of fact = val of num.
constPart ::= where, constDefs,
  env of constPart = env of constDefs.
constPart ::= [],
  env of constPart = emptyEnv.
constDefs ::= constDefs, ',', constDef,
  env of constDefs(1) = env of constDef,
  envh of constDef = env of constDefs(2).
constDefs ::= constDef,
  env of constDefs = env of constDef,
  envh of constDef = emptyEnv.
constDef ::= id, =, exp,
  env of constDef =
     makeEnv(envh of constDef, lex of id, val of exp),
  envh of exp = envh of constDef.
```

Fig. 3. A PAG Attribute Grammar.

```
defun(X+Y,R) :- R is X+Y.
defun(emptyEnv,[]).
defun(makeEnv(Env,Id,Val),[(Id,Val)|Env]).
defun(valueOf(Id,Env),Val) :-
   member((Id,Val),Env),!.
```

Fig. 4. Definition of semantic functions for the attribute grammar of Fig. 3.

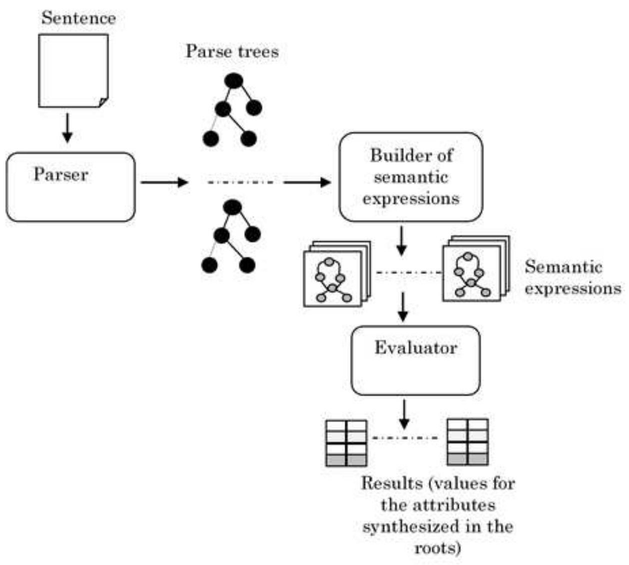

Sentence
Parse trees
Parser
Builder of semantic expressions
Semantic expressions
Evaluator
Results (values for the attributes synthesized in the roots)

Fig. 5. Structure of prototypes built with PAG.

- The *parser* parses sequences of tokens into parse trees.
- The *builder of semantic expressions* traverses these trees and associates a suitable *semantic expression* with each attribute in each node. Semantic expressions are ground terms on the signature of semantic functions, and they will be used to compute the attribute values.
- The *evaluator* component performs the evaluation of the semantic expressions. Actually, it is only needed to evaluate the expressions attached to the synthesized attributes of the parse tree's root.

Notice that the evaluation of attributes is further split into two independent stages. The first one, which can be thought of as a *substitution* step in solving semantic equations, is performed by the builder of semantic expressions. The resulting expressions are actually evaluated by the evaluator during the second stage. Also, notice that a sentence can be parsed into several parse trees (this will be the case with ambiguous syntax). In these cases the prototype will non-deterministically yield several results.

These prototypes can be automatically generated from the specifications by using the following PAG predefined components[4]:

- The *parsing kernel*. This component contains the machinery required to parse sentences into parse trees.
- The *evaluation kernel*. This component is used to evaluate the expressions attached with the semantic attributes.
- The *generator*. This component translates the attribute grammar specification into several working components required to produce the final pro-

[4] From an implementation viewpoint, in the context of this paper *components* are constituted by a set of clauses and optionally, directives to the underlying Prolog engine.

totype.

- The *prelude*. As said before, this component defines several semantic functions that can be reused in different specifications.

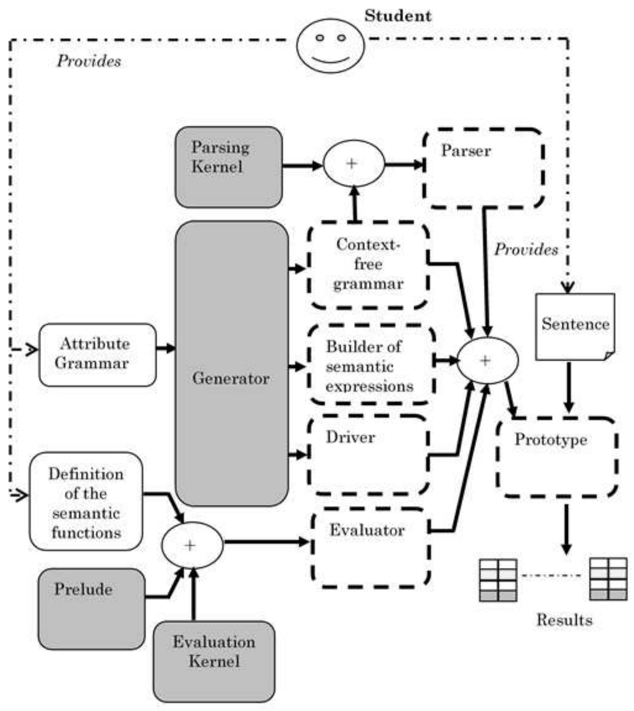


Fig. 6. Prototyping workflow in PAG. Predefined components are shadowed. Generated components are dash-lined. With + we denote unions of clause sets.

The entire process is depicted in Fig. 6, which also highlights the different components in the framework. That way, the process begins when the student specifies the prototype, providing an attribute grammar and defining the semantic functions used. The union of these semantic functions, the prelude and the evaluation kernel yields the prototype's evaluator. In turn, the attribute grammar specification is processed by the generator to produce:

- A description of the underlying context-free grammar that, added to the parsing kernel, will yield the parser component.
- A builder of semantic expressions for the final prototype.
- A driver that will glue all the prototype components together. This driver will be used by the student to run the prototype.

4 Implementing the Prototyping Framework

PAG is based on two simple principles to provide students with the expressive freedom required during prototyping:

- On one hand, the framework is able to process arbitrary context-free grammars. This is carried out by using a suitable implementation of Early's algorithm [6] in the parsing kernel, a general parsing method able to perform a reasonably efficient parsing of sentences regarding any (even ambiguous) context-free grammar.

- On the other hand, the system uses a simple technique for dealing with any non-circular attribute grammar. By interpreting semantic functions in the Herbrand domain (i.e. by interpreting them as term constructors) a logical one-pass attribute grammar (in the sense of [19]) is obtained. Indeed, the builder of semantic expressions can be considered a one-pass evaluator for such a grammar. The semantic expressions produced are subsequently evaluated by the evaluator. Therefore, instead of interleaving parsing, tree traversal, and evaluation, they are kept as separated processes. Separation of parsing and evaluation has been also proposed in [3], where lambda calculus is taken as a notation for semantic expressions and circular dependencies are transformed into lambda calculus fixpoint computations. It has also been proposed in the context of definite clause translation grammars (DCTGs) [1], where parsing yields parse trees decorated with Horn-like semantic rules. PAG also separates parsing and tree traversal to enable arbitrary context-free syntax. In addition, a simple technique is used to avoid reevaluation of common subexpressions in the semantic expressions produced.

The following subsections explore the implementation details. Subsection 4.1 presents the parsing kernel. Subsection 4.2 describes how this kernel is specialized in particular grammars to yield parsers for these grammars. Subsection 4.3 describes the pattern followed by the builders of semantic expressions. Subsection 4.4 describes the implementation of the evaluation kernel. Finally, subsection 4.5 outlines the implementation of the generator.

4.1 The Parsing Kernel

As aforementioned, the parsing kernel is based on Early's algorithm [6]. The algorithm works for arbitrary (even ambiguous) context-free grammars and sentences of length n with a worst-case time complexity in $\mathcal{O}(n^3)$ and space complexity in $\mathcal{O}(n^2)$. Since test sentences are usually small, these overheads are acceptable. In addition, the algorithm is simple and intuitive enough to be easily comprehended and traced by the students, therefore letting them debug the syntax. In the following, we briefly summarize the main aspects of the algorithm and of our implementation.

The central concept in Early's algorithm is that of an *item*. An item is an object of the form $< i, j, X ::= \alpha.\beta >$, indicating an intended situation where:

(i) The parser is at position i on the input.

(ii) The production $X ::= \alpha\beta$ is being used to analyze an input fragment starting at position j.

(iii) An α structure has already been recognized, and the parser is waiting for a β structure.

---

init: $< 1, 1, S' ::= .S, [\ ] >\in \mathcal{I}_G$

closure: $$\frac{<i,j,X::=\alpha.Y\beta,\tau>\in\mathcal{I}_G\ ;;\ Y::=\gamma\in\mathcal{P}_G}{<i,i,Y::=.\gamma,[\ ]>\in\mathcal{I}_G}$$

shift: $$\frac{<i,j,X::=\alpha.a\beta,\tau>\in\mathcal{I}_G\ ;;\ w_i=a}{<i+1,j,X::=\alpha a.\beta,append(\tau,[a])>\in\mathcal{I}_G}$$

reduce: $$\frac{<i,j,X::=\gamma.,\tau>\in\mathcal{I}_G\ ;;\ <j,k,Y::=\alpha.X\beta,\tau'>\in\mathcal{I}_G}{<i,k,Y::=\alpha X.\beta,append(\tau',[t(X,\tau)])>\in\mathcal{I}_G}$$

Fig. 7. Rules characterizing the set of Early's items $\mathcal{I}_G$ for a context-free grammar G with a set of productions $\mathcal{P}_G$ and for a sentence w.

The rules in Fig. 7 characterize all the possible items for a context-free grammar and a sentence. In this characterization we have also enriched items with a fourth component to yield objects with the form $< i, j, X ::= \alpha.\beta, \tau >$. Here τ is the sequence of the parse trees corresponding to the parsed symbols α. Parse trees are represented as terms with the form `t(`*`root`*`, [`*`Child`*$_1$`, ..., `*`Child`*$_k$`])`.

The intended meanings of the rules are:

- The *init* rule establishes the parser initialization. In this rule S denotes the original grammar's axiom, while S' is the new axiom of the grammar expanded with a new production $S' ::= S$. Therefore, the item $< 1, 1, S' ::= .S, [\ ] >$ means that the parser is waiting to recognize the entire input according to the grammar given.
- When waiting for a non-terminal, the *closure* rule makes it possible to *activate* all the productions for this non-terminal.
- The *shift* rule allows the recognition of a terminal on the input.
- The *reduce* rule enables all the rules waiting for a non-terminal to advance when a production for this non-terminal has been finished.

Items of the form $< n + 1, 1, S' ::= S., [t] >$ represent complete parses of the input, with t the corresponding parse tree. Notice that items can be grouped by the input position to yield *parser lists*. For an input of length n, there will be $n + 1$ such lists. Early's algorithm proceeds by:

- Initializing the first list by applying the *init* rule.

- Applying the *closure* and *reduce* rules to the items in each list until reaching an equilibrium.
- Moving to the next list by applying the shift rule.

In addition, by fusing items with a common core (i.e. with the same three first elements) into a single one and by making smart use of *pointers* to kept track of the parse trees, it is possible to overcome the potential exponential complexity of a naïve implementation based on the rules of Fig. 7.[5] Also, we use lookahead information to achieve further improvements in efficiency.

4.2 Specializing the Parsing Kernel

As mentioned in the previous section, to produce a parser for a concrete grammar, a suitable description of this grammar must be added to the parsing kernel. This description includes:

- The grammar's axiom. This is indicated with an `axiom/1` predicate.
- A description of each production. This is indicated with a `prod/3` predicate. The first argument of `prod/3` is a unique identifier for the production. The second argument is the production head. The third argument is the sequence of symbols in the production's body. These symbols are encapsulated in a term with a `body` functor to make access to the arguments in constant time possible. The empty phrase λ is represented with a `body` constant.

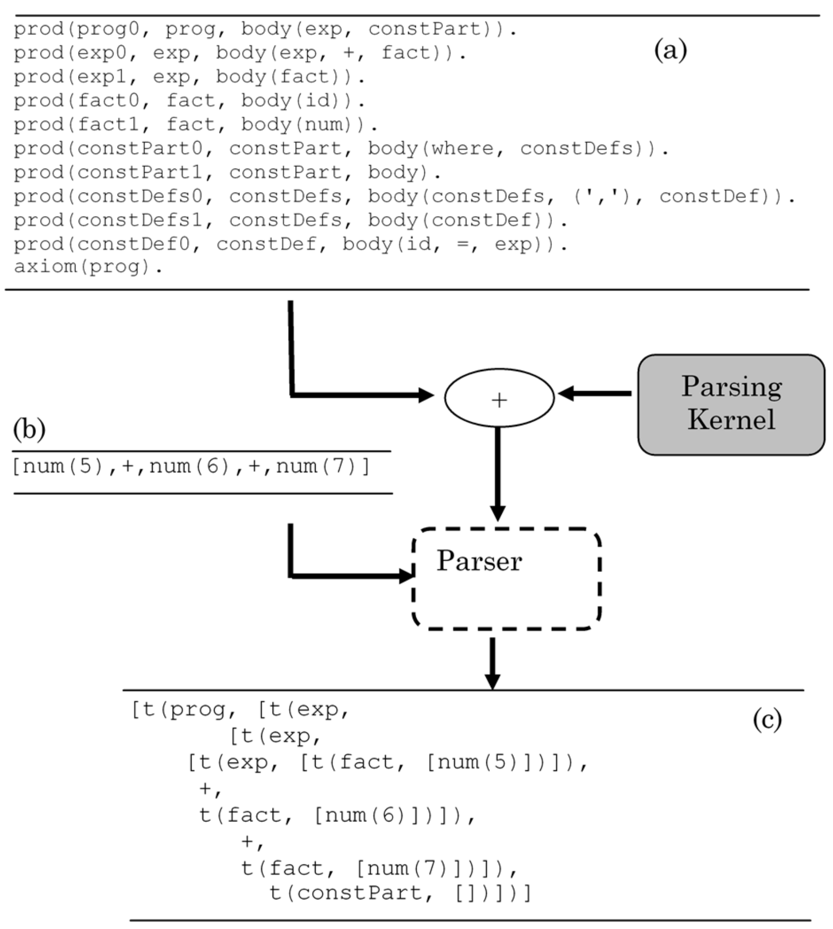
The resulting parsers operate on lists of tokens. While it is possible to attach lexical attributes to these tokens, representing them as terms, only their functors are considered during shifting.

Example 4.1 Fig. 8a shows a representation of the underlying context-free grammar in Fig. 3 to be used with the parsing kernel. The combination of these facts and the parsing kernel yields the parser, as depicted in Fig. 8. This parser can be applied to sentences, as represented in Fig. 8b, in order to yield parse trees with the format illustrated in Fig. 8c.

4.3 Pattern for the Builders of Semantic Expressions

Builders of semantic expressions operate on the parse trees and take full advantage of the unification mechanism in logic programming to automatically solve the dependencies between attributes during a single top-down, left-to-right traversal. As mentioned before, they can be conceived as straightforward implementations of evaluators for non-circular attribute grammars where all the semantic functions have been interpreted as term constructors. These components are structured according to the following common pattern:

[5] Consider a grammar like `A ::=` λ| `aA` | `Aa` with a naïve implementation.



```
prod(prog0, prog, body(exp, constPart)).
prod(exp0, exp, body(exp, +, fact)).
prod(exp1, exp, body(fact)).
prod(fact0, fact, body(id)).
prod(fact1, fact, body(num)).
prod(constPart0, constPart, body(where, constDefs)).
prod(constPart1, constPart, body).
prod(constDefs0, constDefs, body(constDefs, (','), constDef)).
prod(constDefs1, constDefs, body(constDef)).
prod(constDef0, constDef, body(id, =, exp)).
axiom(prog).
```

```
[num(5),+,num(6),+,num(7)]
```

```
[t(prog, [t(exp,
          [t(exp,
      [t(exp, [t(fact, [num(5)])]),
       +,
       t(fact, [num(6)])]),
           +,
           t(fact, [num(7)])]),
             t(constPart, [])])]
```

Fig. 8. (a) Description of the underlying context-free grammar in Fig. 3 for the parsing kernel; (b) a sentence to be processed by the resulting parser; (c) list with the single parse tree associated with this sentence.

- Each non-terminal yields a predicate. The last argument for this predicate corresponds to the parse tree. The other arguments correspond to the semantic attributes.
- The predicate for a non-terminal is defined with a clause for each grammar rule. The body is formed by following the right side of the production.
- Each non-terminal in the body is translated into an invocation to the corresponding predicate. In this invocation, the last argument is bound to the corresponding child in the parse tree. In addition, fresh variables are introduced for each semantic attribute.
- Each terminal is translated by binding the corresponding child to a suitable term, with a fresh variable for each lexical attribute.
- Each copy equation $a\ of\ s = a'\ of\ s'$ is translated as $X = Y$, where X is the variable for $a\ of\ s$, and Y that for $a'\ of\ s'$.
- Any other equation $a\ of\ s = t$ is translated as $X = \#(\_, t')$. X is the variable associated with $a\ of\ s$, and in t all the references to attributes

are substituted by the corresponding variables to yield t'. Furthermore a fresh variable is associated with the resulting term. This variable is called a backup variable, and it will be used to avoid redundant re-evaluations, as indicated in the next subsection.

The resulting builders traverse the parse trees in a depth-first, left-to-right way, binding the attribute variables to their possibly incomplete semantic expressions. Indeed, when a variable associated to an attribute with right-dependencies of other attributes is bound, variables associated with such attributes remain unbound until they are reached. Then unification will fill the holes for free. The pattern works for general non-circular attribute grammars.

```
prog(A, t(prog, [B, C])) :-
        exp(D, E, B),constPart(F, C),A=E,D=F.
exp(A, B, t(exp, [C, D, E])) :-
        exp(F, G, C),D= +,fact(H, I, E),B= #(J, G+I),F=A,H=A.
exp(A, B, t(exp, [C])) :-
        fact(D, E, C),B=E,D=A.
fact(A, B, t(fact, [C])) :-
        C=id(D),B= #(E, valueOf(D, A)).
fact(A, B, t(fact, [C])) :-
        C=num(D),B=D.
constPart(A, t(constPart, [B, C])) :-
        B=where,constDefs(D, C),A=D.
constPart(A, t(constPart, [])) :-
        A= #(B, emptyEnv).
constDefs(A, t(constDefs, [B, C, D])) :-
        constDefs(E, B),C= (','),constDef(F, G, D),A=G,F=E.
constDefs(A, t(constDefs, [B])) :-
        constDef(C, D, B),A=D,C= #(E, emptyEnv).
constDef(A, B, t(constDef, [C, D, E])) :-
        C=id(F),D= (=),exp(G, H, E),
                  B= #(I, makeEnv(A, F, H)),G=A.
```

Fig. 9. The builder of semantic expressions for the grammar in Fig. 3 such as it is automatically generated in PAG.

Example 4.2 Fig. 9 shows the builder of semantic expressions generated by PAG from the specification in Fig. 3.

4.4 The Evaluation Kernel

The evaluation kernel, whose code is shown in Fig. 10, evaluates semantic expressions in an applicative order, with the exception of those affecting non strict functions (for them, the arguments are passed to the function without being evaluated, allowing the customization of any other suitable evaluation strategy). The kernel invokes the semantic functions when defined, or otherwise uses the functors as term constructors. The only tricky aspect of this process is the use of backup variables to avoid the reevaluation of terms duplicated in the expression. Indeed, when an expression of the form $\#(V,E)$ is evaluated:

- If V is free, the expression E is actually evaluated and V is bound to the resulting value.
- If not, the backed up value is used instead.

```
eval(#(Val,Exp),Val) :-
  var(Val),!,
  eval(Exp,Val).
eval(#(Val,_),Val) :- !.
eval(Exp,Val) :-
  nonstrict(Exp),!,
  doResult(Exp,Val).
eval(Exp,Val) :-
  Exp =.. [F|Args],
  evalArgs(Args,VArgs),
  Funcall =.. [F|VArgs],
  doResult(Funcall,Val).
evalArgs([],[]).
evalArgs([Exp|Exps],[Val|Vals]) :-
  eval(Exp,Val),
  evalArgs(Exps,Vals).
doResult(Funcall,Val) :-
  defun(Funcall,Val),!.
doResult(Funcall,Funcall).
```

Fig. 10. The evaluation kernel.

```
#(A,
   #(B, valueOf(x,
     #(C,
        makeEnv(#(D, emptyEnv),
                      x, 5))))
     +
   #(E, valueOf(x,
     #(C,
        makeEnv(#(D, emptyEnv),
                      x, 5)))))
```

Fig. 11. A semantic expression with duplicated subexpressions.

Example 4.3 In Fig. 11 the semantic expression for the `val` attribute of `prog` and for the input `[ id(x), +, id(x), where, id(x), =, num(5) ]` is shown. Notice that in this expression the subexpression for looking up the value of `x` is duplicated. Nevertheless, each duplicated expression is only evaluated once, since all the duplicates share the same backup variable. Also notice that the term shown in Fig. 11 is an externalization of the corresponding structure, which can be stored efficiently by sharing common substructures.

4.5 The Generator

The generator processes the attribute grammar specification to produce a context-free grammar description, a builder of semantic expressions, and a driver. The grammar description and the parsing kernel yield the parser, which is connected to the builder and the evaluator in the cited driver.

All the elements in the specifications, including the rules, are treated as Prolog facts by the generator. Indeed, the syntax of the specification language is easily embedded in Prolog by properly defining the `::=` and the `of` operators. The generator can be integrated with the Prolog system by using the static metaprogramming facilities found in many Prolog implementations. This lets students directly load PAG specifications into the Prolog engine.

5 Conclusions and Future Work

In this paper we have presented PAG, a framework for the rapid prototyping of language processors in Prolog. This framework is oriented to supporting the learning process of students enrolled in an introductory course in language processors by letting them test their specifications. The framework is able to deal with general non-circular attribute grammars on arbitrary (maybe ambiguous) context-free syntax in a comprehensible way, which is a primary requirement in the application context mentioned. To deal with arbitrary syntax, Early's parsing algorithm is used. General non-circular attribute grammars are managed by first interpreting semantic functions as term constructors. The semantic expressions yielded are then definitively evaluated considering the actual definitions for the semantic functions. The overhead incurred by the cited separation of concerns is acceptable in a prototyping context, where the simplicity and comprehensibility of the techniques for the average student come before considerations of efficiency.

Currently we are extending PAG with simple modularization facilities based in the composition of *semantic aspects*, as suggested in [13]. This is in accordance with our pedagogical method, since we introduce different *views* of the complete attribute grammar: one for the construction of the symbol table, another for checking the contextual constraints on the source language, and a third one for dealing with the translation concerns. We are also considering the extension of PAG to deal with circular attribute grammars. This extension is oriented to our students of a Ph.D. course on e-learning, where we promote the use of language processor technologies in the processing of the markup languages proposed by the different e-learning specifications (see, for instance, [16]). The basic idea is to work with circular Prolog terms in managing circular definitions. With this we hope to provide students, who belongs to several disciplines, with a less knowledge-demanding alternative than the one based on fixpoint computations [3]. We are also planning to include domain-specific visual tracing capabilities in the system. As future work we want to use PAG in an introductory course on computational linguistics. We also want to take advantage of the modularity of the approach to build a complete learning scenario based on the learning object paradigm and supported by the web-based e-learning systems deployed at our university.

References

[1] Abramson, H. Dahl, V, "Logic Grammar", Springer, 1989.

[2] Aho, A. Sethi, R. Ullman, J. D, "Compilers: Principles, Techniques and Tools", Adisson-Wesley, 1986.

[3] Arbab, B, *Compiling Circular Attribute Grammars into Prolog*, IBM Journal of Research and Development **30**(3) (1986), 294–309.

[4] Carpenter, B. Penn, G, "The Attribute Logic Engine User's Guide Version 3.2.1", University of Toronto, 2001.

[5] Clocksin, W. F. Mellish, C. S, "Programming in Prolog", Springer, 1987.

[6] Early, J, *An Efficient Context-free Parsing Algorithm*, Communications of the ACM **13**(2) (1970), 94–102.

[7] "Focus on the Structure of Higher Education in Europe 2004/05. National Trends in the Bologna Process", Eurydice, 2005.

[8] Fisher, C. N. LeBlanc, R. J. J, "Crafting a Compiler", The Benjamin/Cummings Publishing Company, 1988.

[9] Gray, R. W. Heuring, V. P. Levi, S. P. Sloane, A. M. Waite, W. M, *Eli: A Complete, Flexible Compiler Construction System*, Communications of the ACM **35** (1992), 121–131.

[10] Grosch, J. Emmelmann, H, "A Tool Box for Compiler Construction", CoCoLab White Paper, 1990.

[11] Jansen, P. Augusteijn, L. Munk, H, "An Introduction to Elegant. Second Edition", Philips Research Laboratories, 1993.

[12] Jourdan, M. Parigot, D, *Internals and Externals of the FNC-2 Attribute Grammar System*, in Ablas, H. Melichar, B. (eds), "Attribute Grammars, Applications and Systems", Lecture Notes in Computer Science **545**, Springer, 1991.

[13] Kastens, U, *Attribute Grammars as an Specification Method*, in Ablas, H. Melichar, B. (eds), "Attribute Grammars, Applications and Systems", Lecture Notes in Computer Science **545**, Springer, 1991.

[14] Klint, P. Lämmel, R. Verhoef, C, *Toward an Engineering Discipline for Grammarware*, ACM Transactions on Software Engineering and Methodology **14**(3) (2005), 331–380.

[15] Knuth, D. E, *Semantics of Context-free Languages*, Mathematical Systems Theory **2**(2) (1968), 127–145. See also the correction published in Mathermatical System Theory **5** (1) (1971), 95–96.

[16] Koper, R. Tatersall, C (eds.), " Learning Design, a Handbook on Modelling and Delivering Networked Education and Training", Springer, 2005.

[17] Mernik, M. Lenič, M. Avdičauševič E. Žumer, V, *Compiler/Interpreter Generator System LISA*, Proc. of the 33rd Hawaii International Conference on Systems Science, 2000.

[18] Mernik, M. Žumer, V, *An Educational Tool for Teaching Compiler Construction*, IEEE Transactions on Education **46**(1) (2003), 61–68.

[19] Paakki, J, *A Logic Based Modification of Attribute Grammars for Practical Compiler Writing*, Proc. of the 7h International Conference on Logic Programming, 1990.

[20] Paakki, J, *Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation*, ACM Computing Surveys **27**(2) (1995), 196–255.

[21] Polsani, P. R, *Use and Abuse of Reusable Learning Objects*, Journal of Digital Information, **3**(4), 2003.

[22] Rehberg, S. Ferguson, D. McQuillan, J. Eneman, S. Stanton, L, "The Ultimate WebCT Handbook, a Practical and Pedagogical Guide to WebCT 4.x", Ultimate Handbooks, 2004

[23] Sierra, J. L. Fernández-Valmayor, A. Guinea, M. Hernanz, H. Navarro, A, *Building Repositories of Learning Objects in Specialized Domains: The Chasqui Approach*, Proc. of the 5th IEEE International Conference on Advanced Learning Technologies ICALT05, 2005.

[24] Sterling, L. Shapiro, E, "The Art of Prolog", The MIT Press, 1994.

[25] Šveda, M. Jankovský,M, *Prototyping with Attribute Grammars and Prolog*, Proc. of the 23rd EUROMICRO Conference'97 New Frontiers of Information Technology, 1997.

[26] Warren, D. S, "Programming in Tabled Prolog (Draft)", 1999, Available online at http://www.cs.sunysb.edu/∼warren/xsbbook/book.html (last visited on Febraury 9, 2006).

AspectLISA: an aspect-oriented compiler construction system based on attribute grammars

Damijan Rebernak, Marjan Mernik [1,2]

University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova ul. 17, 2000 Maribor, Slovenia

Pedro Rangel Henriques [3]

University of Minho
Department of Computer Science
Campus de Gualtar
4710 - 057 Braga, Portugal

Maria João Varanda Pereira [4]

Polytechnic Institute of Bragança
Campus de Sta. Apolónia
Apartado 134 - 5301-857, Bragança, Portugal

Abstract

The use of object-oriented techniques and concepts, like encapsulation and inheritance, greatly improves language specifications towards better modularity, reusability and extensibility. Additional improvements can be achieved with aspect-oriented techniques since semantic aspects also crosscut many language constructs. Indeed, aspect-oriented constructs have been already added to some language specifications.

The LISA compiler construction system follows an object-oriented approach and has already implemented mechanisms for inheritance, modularity and extensibility. Adding aspects to LISA will lead to more reusable language specifications. In the paper, aspect-oriented attribute grammars are introduced, and the underlying ideas are incorporated into AspectLISA, an aspect-oriented compiler generator based on attribute grammars.

Key words: Attribute grammars, aspect-oriented programming, compiler generators.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

The challenge in programming language definition is to support modularity and abstraction in a manner that supports reusability and extensibility. A language designer often wants to include new language features incrementally as the programming language evolves. This is especially true in developing domain-specific languages (DSLs) which change more frequently than general-purpose programming languages [17]. Ideally, a language designer would like to build a language simply by reusing different language definition modules (e.g., language components), such as modules for expressions, declarations, etc., regardless of the different formal methods that may be used to specify such language components. This approach is common in component-based programming [24] where components can be simply plug-ins.

This cannot be done now, even if we restrict ourselves to just one of the formal methods (abstract state machines, action semantics, algebraic specifications, attribute grammars, denotational semantics, operational semantics, two-level grammars, etc. [23]) since different compiler-compilers (automatic compiler generation systems) use different and incompatible specification languages (e.g., despite the fact that Eli [6] and FNC-2 [9] both rely on attribute grammars one can not exchange language definition modules written in the other system). Moreover, the same is usually true even in the case of the same specification language since syntax entities (e.g., non-terminals and terminals) and semantic entities (e.g., attributes and semantic rules in the case of attribute grammars) are not constituents of the hidden part of the module, nor are the parameters of language definition modules. For example, when importing a module for expressions some non-terminals may clash with existing non-hidden non-terminals producing undesirable effects. Such a module can be parameterized using non-terminals as parameters to solve renaming problems. However, modules with dozens of parameters are hard to use.

Compared to modern programming languages, such as object-oriented or functional languages, language specifications of the 1980's and early 1990's were far less advanced, specifically concerning provisions for abstraction, modularization, extensibility and reusability. Recently, concepts from general programming languages have been successfully incorporated into language specifications. Among them, object-oriented techniques are one of the most successful. Indeed, this had several benefits on language specifications. To fully achieve modularity, extensibility and reusability these techniques need to be combined with aspect-oriented techniques because semantic aspects also crosscut many language constructs [19]. These observations have been taken into account in extending the LISA specification language [18] with aspect-oriented features. The paper presents AspectLISA,

[1] This work is sponsored by bilateral project BI–PT/04–06–008 "Grammar Based Systems" between Slovenia and Portugal.

[2] Email: {`damijan.rebernak, marjan.mernik`}`@uni-mb.si`

[3] Email: `prh@di.uminho.pt`

[4] Email: `mjoao@ipb.pt`

which is an aspect-oriented compiler generator based on attribute grammars.

The paper is organized as follows. A brief introduction to aspect-oriented programming is given in section 2 followed by related work presented in section 3. The main part of the paper constitutes section 4 where AspectLISA is discussed. A small case study illustrating ideas is given in section 5. The concluding comments are mentioned in section 6.

2 Aspect-oriented programming

The major abstraction technique in software engineering is to divide the system into functional components in such manner that changes to a particular component do not propagate through the entire system [5,22]. However, some issues, called aspects, are system wide and cannot be put into a single functional component. As examples, failure handling, persistence, communication, coordination, memory management, are aspects of a system behavior that tend to crosscut groups of functional components. As a consequence, functional components are tangled with aspect code. This tangling problem makes functional components less reusable and difficult to develop, understand, and evolve. A solution is provided by aspect-oriented programming (AOP) [12] which is a programming technique for modularizing concerns that crosscut the basic functionality of programs. In AOP, aspect languages are used to describe properties which crosscut basic functionality in a clean and a modular way. Despite that the main part of AOP research is devoted to general-purpose languages [11,16] similar problems exists in domain-specific languages. For example, in language specifications modularization is usually based on language syntax constructs, whereas the modularization based on different aspects (e.g. name analysis, type checking, code generation, etc.) would be more beneficial. To overcome this problem aspect-oriented techniques can be used.

In order to achieve the desired properties of the system, we need an aspect weaver that combines the component and the aspect language by weaving advice at appropriate join points and may involve merging components, modifying and optimizing them.

3 Aspects in language development

Aspect-oriented programming is a very promising approach and has been successfully used in tools for language definition and implementation [4,7,10,13,15,26,27]. In this context, aspects have been used for many different tasks (e.g., in [26] an extension for weaving debugging information into DSL specifications is reported). In the rest of this section we describe in more detail some of the more relevant contributions in the field, *using aspects in language specification or implementation*.

3.1 JastAdd

JastAdd [7] is a Java-based system for compiler construction. JastAdd is centered around object-oriented representation of the abstract syntax tree (AST). Nonterminals act as abstract super classes and productions act as specialized concrete subclasses that specify the syntactic structure, attributes and semantic rules. All these elements can be inherited, specialized, and overridden in subclasses. The idea of aspect-orientation in JastAdd is to define each aspect of the language in a separate class and then weave them together at appropriate places. The JastAdd system is a class weaver: it reads all the JastAdd modules and weaves the fields and methods into the appropriate classes during the generation of the AST classes. With separation of different language aspects among different classes, developers have the possibility to use all features of Java to specify aspects. In the following example, taken from [7], two different aspects are described in separate classes. The first one (`typechecker.jadd`) performs type checking for expressions and computes the boolean field `typeError`. The `unparser.jadd` (second example) implements an unparser which makes use of the field `typeError` to report type-checking errors.

```
// typechecker.jadd
class Exp {
 abstract void typeCheck(String expectedType);
}
class Add {
   boolean typeError;
 void typeCheck(String expectedType) {
   getExp1().typeCheck("int");
    getExp2().typeCheck("int");
    typeError= expectedType != "int";
  }
}

// unparser.jadd
import Display;
class Stmt {
  abstract void unparse (Display d);
}
class Add {
  void unparse (Display d) {
    ...
    if (typeError)
      d.showError("type mismatch");
    }
}
```

Every field, method, import declaration is weaved to all generated AST classes, as can be seen in following example.

```
 class ASTAdd extends ASTExp {
  // Access interface
  ASTExp getExp1() { ... }
  ASTExp getExp2() { ...}
  // From typechecker.jadd
  boolean typeError;
  void typeCheck(String expectedType) {
    getExp1().typeCheck("int");
    getExp2().typeCheck("int");
    typeError = expectedType != "int";
  }
  // From unparser.jadd
  void unparse(Display d) {
    ...
    if (typeError)
      d.showError("type mismatch");
    ...
  }
}
```

As can be seen in the example above, this approach does not follow the conventional AOP join point model (JPM) where join points are specified using a pointcut pattern language. However, it can be seen as inter-type declarations in AspectJ [11] where join points are all non-anonymous types in the program and pointcuts are the names of classes or interfaces.

3.2 *AspectG*

To generate an additional language-based tool (e.g., debugger) new specifications need to be added in several places in language specifications [8]. These new additions can be seen as aspects (e.g., debugging aspects). It was observed [26] that such aspects crosscut basic language specifications. Hence, the aspect-oriented language AspectG [2] was created for modular implementation of crosscutting concerns in the ANTLR language definition [1]. Since ANTLR belongs to syntax directed translations (semantic rules are not declaratively specified and order of semantic rules is important) AspectG uses the following model:

- join points are static points in language specifications where additional aspects can be weaved,
- pointcuts specify join points and include not only the syntax level of the grammar but also the semantics associated with a particular syntax (see within and match constructs in the example below),
- advice are similar to AspectJ notion (before and after) and brings together a pointcut and a body of code.

An example of pointcut and advice in AspectG is shown below.

```
 ...
 command
  :( RIGHT
    {
     fileio.print("//move right");
     fileio.print("x=x+1;");
     fileio.print("time=time+1;");
   }
 ...

pointcut count_gpllinenumber():
  within(command.*) &&
  match (fileio.print("x=x+1;"));

 after(): count_gpllinenumber()
  {gplbeginline=fileio.getLinenumber();
   gplendline=fileio.getLinenumber();}
```

The result of weaving is:

```
...
command
 :( RIGHT
   {
    fileio.print("//move right");
    fileio.print("x=x+1;");
    gplbeginline=fileio.getLinenumber();
    gplendline=fileio.getLinenumber();
    fileio.print("time=time+1;");
  }
...
```

3.3 *AspectASF*

AspectASF [13] is a simple aspect language for language specifications written in the ASF+SDF [25] formalism. Only rewrite rules are supported. Therefore, join points in AspectASF are static points in equation rules describing semantics of the language. The pointcut pattern language in AspectASF is a very simple pattern matching language on the structure of equations where only labels and left-hand sides of equations can be matched. Pointcuts can be of two types: entering an equation (after a succesfull match of left-hand side) and exiting an equation (just before returning the right-hand side). Examples (all examples in this section are taken from [13]) of pointcuts in AspectASF language are:

`[_]`	matches all equations
`[_] eval (_, _)`	matches all equations with outermost symbol eval
`[_] eval (_, Env)`	matches all equations with 2nd arg an Env variable
`[int*] _ or [real*] _`	matches all equations with label int.. or real..

Advice code specify additional equations which are written in the ASF formalism. There are two types of advice: after entering an equation (concatenating equations to the beginning of the list of equations that is matched by the pointcut) and before exiting an equation (concatenating equations to the end of the list of equations that is matched by the pointcut). An example of AspectASF is shown below.

```
[1] Env'  := evs(Stat, Env),
    Env'' := evs(Stat*, Env')
    ==============================
    evs(Stat ; Stat*, Env) = Env''

pointcut statementStep: entering [_] evs(Stat ; Stat*, Env)
after: statementStep tide-step(get-location(Stat))
```

After weaving takes place the aspects are weaved into the original language specifications. In other words, additional equations are appended to appropriate places.

```
[1] tide-step(get-location(Stat)),
    Env'  := evs(Stat, Env),
    Env'' := evs(Stat*, Env')
    ==============================
    evs(Stat ; Stat*, Env) = Env''
```

4 AspectLISA

4.1 Introduction to LISA

In the LISA project [18,20], one of the main goals was to enable incremental language development. It was soon recognized that inheritance can be very helpful since it is a language mechanism that allows new definitions to be based on the existing ones. A new specification can inherit the properties of its ancestors, and may introduce new properties that extend, modify or defeat its inherited properties. In object-oriented languages the properties that consist of instance variables and methods are subject to modification. The corresponding properties in language definitions based on attribute grammars are:

- lexical regular definitions,
- attribute definitions,
- rules which are generalized syntax rules that encapsulate semantic rules, and
- operations on semantic domains.

Therefore, regular definitions, production rules, attributes, semantic rules and operations on semantic domains can be inherited, specialized or overridden from ancestor specifications. In this approach the attribute grammar as a whole is subject to inheritance employing the “Attribute grammar = Class” paradigm [21].

A very simple language for moving a robot can illustrate our incremental language development approach [20]. The language for robot movement is defined in Fig. 1. The robot can move in different directions and the task is to compute its final position. Over time, the language is extended with new features. For example, we would like to know when the robot will reach the final position. The new language (RobotTime) is specified as an extension of the Robot language (Fig. 2). This is a good example of how different aspects can be modularized in our approach. In the Robot language just the semantic rules for robot movement have been described, while the RobotTime language contains just the semantic rules for time calculation. The RobotTime language inherits regular definitions, syntax constructs and semantic rules from the Robot language and adds new semantic rules for time calculation. Note that the same effect is obtained by implicit pointcuts in aspect-oriented systems like JastAdd [7] (see section 3).

As already mentioned, object-oriented techniques and concepts need to be combined with aspect-oriented techniques to achieve better modularity, extensibility and reusability. This issue is further described in the following sections.

4.2 Aspect-oriented Attribute Grammars

Aspect-oriented attribute grammar (AspectAG) is an attribute grammar [14] extended with pointcut and advice specifications [12], $AspectAG = (G, A, R, Pc, Ad)$. Context-free grammar $G = (N, T, S, P)$, set of attributes A, and set of semantic rules R have the same standard meaning of attribute grammars, as for example described in [18].

```
language Robot {
 lexicon {
   Commands left | right | up | down
   ReservedWord  begin | end
   ignore [\0x0D\0x0A\ ]   // skip whitespaces
 }

 attributes Point *.inp, *.outp;

 rule start {
  START ::= begin COMMANDS end  compute {
    START.outp = COMMANDS.outp;
    // robot position in the beginning
    COMMANDS.inp = new Point(0, 0); };
  }

 rule moves {
  COMMANDS ::= COMMAND COMMANDS  compute {
    COMMANDS[0].outp = COMMANDS[1].outp;  // propagation of position
    COMMAND.inp = COMMANDS[0].inp;        // to sub-commands
    COMMANDS[1].inp = COMMAND.outp; }
  |  epsilon  compute {
    COMMANDS.outp = COMMANDS.inp; };
  }

 rule move {
  // each command changes one coordinate
  COMMAND ::= left  compute {
    COMMAND.outp = new Point((COMMAND.inp).x-1,(COMMAND.inp).y); };
  COMMAND ::= right  compute {
    COMMAND.outp = new Point((COMMAND.inp).x+1,(COMMAND.inp).y); };
  COMMAND ::= up  compute {
    COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y+1); };
  COMMAND ::= down  compute {
    COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y-1); };
  }
}
```

Fig. 1. Robot Language using LISA

Pointcuts Pc is a set of **pointcut productions**, $Pc = \{pc_1, ..., pc_m\}$, where pointcut production pc_i, $1 \leq i \leq m$, has the following form:

$$pc_i < X_1, ..., X_r > \ : \ LHS \ \rightarrow \ RHS$$

In pointcut production pc_i special wildcard symbols (.., $*$) can be used. Wildcard symbol '$*$' denotes a symbol or some part of its name and can be used in the LHS and RHS. Wildcard symbol '..' denotes zero or more symbols and can be used only in the RHS. Symbols $X_i, 1 \leq i \leq r$, are symbols from LHS and RHS and denote the public interface for advice. A pointcut production $pc_i < X_1, ..., X_r >$: $LHS \rightarrow RHS$, selects a production $p : X_0 \rightarrow X_1 ... X_n \in P$ if X_0 matches LHS and X_1 ... X_n match RHS. Let Pm_i denote the set of productions selected by pointcut production pc_i, $Pm_i = \{p_i | p_i \in P \ and \ p_i \ is \ matched \ by \ pc_i\}$. Matched productions Pm selected by pointcuts Pc is then defined as $Pm = \bigcup_{i=1..m} Pm_i$, $Pm \subseteq P$. To match productions Pm, additional semantic rules specified in advice Ad are attached.

Ad is a set of **advice**, $Ad = \{ad_1, ..., ad_l\}$, where advice ad_k, $1 \leq k \leq l$, has

```
language RobotTime extends Robot {

  attributes double *.time;

  rule extends start {
   compute {
      // initial position is inherited
      START.time = COMMANDS.time; }
  }

  rule extends moves {
   COMMANDS ::= COMMAND COMMANDS compute {
     // total time is sum of times spent in sub-commands
     COMMANDS[0].time = COMMAND.time + COMMANDS[1].time; }
   | epsilon  compute {
     COMMANDS.time = 0; };
  }

  rule extends move { // each command spent 1 time step
   COMMAND ::= left compute {
     COMMAND.time = 1; };
   COMMAND ::= right compute {
     COMMAND.time = 1; };
   COMMAND ::= up compute {
     COMMAND.time = 1; };
   COMMAND ::= down compute {
     COMMAND.time = 1; };
  }
}
```

Fig. 2. RobotTime Language using LISA

the following form:

$$ad_k < S_1, ..., S_r > \ on \ pc_i \ \{Rs_k\}$$

Semantic rules Rs_k has the following form:

$$Rs_k = \{S_j.a = f(y_1, ..., y_k) | a \in A(S_j), y_i \in (A(S_1) \cup ... \cup A(S_r)), \ 1 \leq i \leq k\}$$

Defining attributes attached to symbols $S_j, 1 \leq j \leq r$, are defined by semantic rules in Rs_k. Advice ad_k is applied on pointcut pc_i, which match productions Pm_i. For each match production $p_i \in Pm_i$, the actual set of semantic rules Ra_{ki} is obtained by replacing formal symbols S_j (specified in ad_k) by actual symbols X_j (specified in pc_i) in Rs_k. The set of semantic rules Ra obtained from advice Ad and pointcuts Pc is defined as $Ra = \bigcup_{k=1..l, i=1..m} Ra_{ki}$ and needs to be merged with ordinary semantic rules Rp_i, to obtain well defined attribute grammar $AG = (G, A, R')$ in the following manner: $Rp'_i = Rp_i \cup (\bigcup_{k=1..l} Ra_{ki})$, $R' = \bigcup_{i=1..n} Rp'_i$. Note that $(G, A, R, Pc, Ad) = (G, A, R')$. Therefore, an aspect-oriented attribute grammar is an attribute grammar where some semantic rules are not attached explicitly to production rules but implicitly as advice into productions selected by pointcuts. When semantic rules are merged, only one semantic rule for each defining attribute must exist, otherwise the attribute grammar is not well defined [14]. The following illustrates a simple example:

```
Ordinary attribute grammar specifications:
p0:  A → B C {A.x = B.x + C.x; B.y = 0; C.y = 1;}   // Rp0
p1:  B → a B {B0.x = B1.x; B1.y = B0.y + 1;}        // Rp1
p2:  B → ϵ {B.x = B.y;}                             // Rp2
p3:  C → c {C.x = C.y + 2;}                         // Rp3
Pointcuts:
pc1 <B> :  B → ..                                   // matches p1 and p2
pc2 <A, B> :  A → B *                               // matches p0
Advice:
ad1 <X> on pc1 {X.z=1;}                             // Ra11 = {B.z=1;}
                                                    // Ra12 = {B.z=1;}
ad2 <Y, X> on pc2 {Y.w = X.z;}                      // Ra20 = {A.w = B.z;}
Final semantic rules:
Rp'0  =  Rp0 ∪ Ra20  =  {A.x = B.x + C.x; B.y = 0; C.y = 1; A.w = B.z;}
Rp'1  =  Rp1 ∪ Ra11  =  {B0.x = B1.x; B1.y = B0.y + 1; B.z = 1;}
Rp'2  =  Rp2 ∪ Ra12  =  {B.x = B.y; B.z = 1;}
Rp'3  =  Rp3         =  {C.x = C.y + 2;}
```

4.3 AspectLISA constructs

As seen from Fig. 1 and Fig. 2 LISA enables good modularity and extensibility of attribute grammar specifications. However, there are still situations when new semantic aspects crosscut basic modular structure. In other words, some semantic rules need to be repeated in different productions (e.g., semantic rule `COMMAND.time = 1;` which has to be repeated several times in generalized production `move` of `RobotTime` language). To avoid this unpleasant situation, an aspect-oriented attribute grammar, as specified in subsection 4.2, has been incorporated into LISA language specifications. This extension is called AspectLISA. Join points in AspectLISA are static points in language specifications where additional semantic rules can be attached. These points can be syntactic production rules or generalised LISA rules. The production matching takes place on productions which are members of generalized LISA rules. One pointcut can match productions in different languages over the entire hierarchy of languages. For each pointcut we can define several advice which are parameterized semantic rules written as native Java assignment statements. In AOP, several different approaches of applying aspects to pointcuts exists, like before, after and around [11]. In AspectLISA there is only one way to apply advice on a specific pointcut, since attribute grammars are declarative and the order of equations in semantic rules is not important. Therefore, applying advice before/after a join point is not applicable.

The AspectLISA specification language, including apect-oriented features, pointcuts and advice, has the following parts (note how pointcuts and advice defined in section 4.2 are written in the LISA specification language):

```
language L1 [extends L2, ..., LN] {
   lexicon {
      [[Q] overrides | [Q] extends] R regular expr.
         ⋮
   }
   attributes type At1, ..., AtM
      ⋮
   pointcut P< [S1, ..., Sr] > L.Y : LhsP ::= RhsP ;
      ⋮
   advice [[B] extends | [B] overrides] A< [T1, ..., Tr] > on P {
      semantic functions
   }
      ⋮
   rule [[Y] extends | [Y] overrides] Z {
      X ::= X11 X12 ...  X1p compute {
            semantic functions }
         ⋮
      |
         Xr1 Xr2 ...  Xrt compute {
            semantic functions }
      ;
   }
      ⋮
   method [[N] overrides | [N] extends] M {
      operations on semantic domains
   }
   ⋮
}
```

Symbols used in formal AspectLISA specifications above have following meaning:

- L – language name,
- Q and R – regular expression name,
- At – attribute name,
- P – pointcut name,
- S – actual symbol,
- $LhsP$ and $RhsP$ – left and right-hand side of pointcut production,
- A – advice name,
- T – formal symbol,
- X – grammar symbol,
- Y and Z – grammar rules,
- N and M – method names.

This section focuses only on the new aspect-oriented features of the LISA specification language which are pointcuts and advice.

Pointcuts are defined using the reserved word **`pointcut`**. Each pointcut has a unique name and a list of actual parameters (terminals and non-terminals used in semantic functions of advice). As we already mentioned, join points are static points in language specifications where advice can be applied. In the pointcut definition one can use two wildcards. The wildcard '..' matches zero or more terminal or non-terminal symbols and can be used only to specify right-hand side matching rules. The wildcard '$*$' is used to match parts or whole literal representing a symbol. To illustrate the AspectLISA pointcut model, we present some examples of pointcut specifications, defined over the Robot languages (Fig. 1, Fig. 2).

`*.* :  * ::= ..`	*matches any production in any rule in all languges across current languge hieararchy*
`RobotTime.m* :  * ::= ..`	*matches any production in all rules which start with `m` in `RobotTime` language*
`*.* :  COMM* ::= ..  *D`	*matches all productions in any rule whose left-hand side symbol satisfy pattern "COMM*" and the right-hand side's last symbol ends with `D`*
`Robot.move :  COMMAND ::= left`	*matches only a production `COMMAND ::= left` in the rule `move` of `Robot` language*

Advice in AspectLISA are additional semantics that can be appended at a specific join point. In order to increase reusability, advice are parameterized. Parameters can be terminal or non-terminal symbols and are evaluated at weaving time. Advice are defined using the reserved word **`advice`** and contain information about the pointcut where advice will appear. Below is an example of advice; more examples of advice and pointcuts are provided in section 5.

```
pointcut SimpleCommand<COMMAND> *.move : COMMAND ::= *;
advice SetTime<C> on SimpleCommand { C.time = 1;}
```

The result of weaving advice `SetTime` on pointcut `SimpleCommand` in the RobotTime language is an additional semantic rule `COMMAND.time = 1;` in all productions of rule `Robot.move`. The notation is much simpler as in Fig. 2. The new aspect of the language, namely time calculation, is described at one place (advice) and is not repeated in several productions.

4.4 AspectLISA inheritance

The AspectLISA specification language is an extension of LISA with two new mechanisms (pointcuts and advice). Obviously, pointcuts and advice can also be inherited from ancestor specifications. Formal definition of multiple attribute grammar inheritance as described in [18] needs to be adopted. Due to lack of space in this paper only the formal definition of inheritance of pointcuts and advice are given. For theoretical background and further details readers are referred to [18].

Properties of aspect-oriented attribute grammars consist of lexical regular definitions, attribute definitions, rules which are generalized syntax rules that encapsulate semantic rules, **pointcuts**, **advice** and methods on semantic domains.

$$Property = RegdefName + AttributeName + RuleName + PointcutName + AdviceName + MethodName$$

For each pointcut pc in the language l, a $Pointcuts(l)(pc)$ is a finite set of matching productions P, that match to the pointcut pc, over the hierarchy of language l.

$$Pointcuts : Language \rightarrow PointcutName \rightarrow MatchingProductionRules$$
$$Pointcuts(l)(pc) = \{p_i \mid p_i \in P, p_i : X_{i0} \rightarrow X_{i1}X_{i2}...X_{in}, match(p_i, pc)\}$$

For each advice ad attached to pointcut pc in the language l, $Advice(l)(ad)(pc)$ is a finite set ($ProdSem$) of pairs (p, R_p), where p is a production and R_p is a union of finite set of semantic rules associated with the production p, and semantic rules ($definedR_p$) defined by advice ad, where formal symbols of advice are replaced by actual symbols defined in pointcut pc.

$$Advice : Language \rightarrow AdviceName \rightarrow PointcutName \rightarrow ProdSem$$
$$Advice(l)(ad)(pc) = \{(p, R_p) | p \in Pointcuts(l)(pc),$$
$$p : X_0 \rightarrow X_1X_2...X_n,$$
$$R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) | \ X_i.a \in DefAttr(p)\} \cup definedR_p(ad, pc)\}$$

Multiple aspect-oriented attribute grammar inheritance is defined as follows.

Let $AspectAG_1$, $AspectAG_2$, ..., $AspectAG_m$ be aspect-oriented attribute grammars formally defined as:

$$AspectAG_1 = (G_1, A_1, R_1, Pc_1, Ad_1),$$
$$AspectAG_2 = (G_2, A_2, R_2, Pc_2, Ad_2),$$
$$\vdots$$
$$AspectAG_m = (G_m, A_m, R_m, Pc_m, Ad_m), \text{ then}$$

$$AspectAG = AspectAG_2 \oplus \ldots \oplus AspectAG_m \oplus \triangle AspectAG_1 ,$$

where $AspectAG_1$, which inherits from $AspectAG_2, \ldots, AspectAG_m$, is defined as:

$$AspectAG = (G, A, R, Pc, Ad), \text{ where}$$
$$\begin{aligned} G &= G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1, \\ A &= A_1 \ominus \ldots \ominus A_m, \\ R &= R_1 \otimes \ldots \otimes R_m, \\ Pc &= Pc_1 \ominus \ldots \ominus Pc_m, \\ Ad &= Ad_1 \otimes \ldots \otimes Ad_m . \end{aligned}$$

Therefore, inheritance on pointcuts is defined in a similar manner as for attributes [18]. Pointcuts as well as attributes cannot be extended, but can be inherited from ancestor attribute grammars. On the other hand, it is possible that some pointcut is redefined in current specifications which override pointcut specified in ancestor specifications. Inheritance on advice is defined in a similar manner as for semantic rules R [18]. This should not be surprising, because advice are just additional semantic rules which need to be weaved at appropriate join points.

4.5 AspectLISA novelty

AspectLISA is first specification language based on attribute grammars that use an explict pointcut model. Note that in JastAdd the pointcut model is implicit. The poincut model in AspectG is more complicated because syntax as well as semantic level are involved in the specification. This is due to using syntax directed translation instead of attribute grammars. AspectASF uses very simple pattern matching language where only labels and left-hand sides of equations written in ASF formalism can be matched. None of the existing systems enable inheritance on advice and pointcuts. Moreover, advice in AspectLISA are parameterized on grammar symbols and hence more reusable.

5 Using AspectLISA

Each LISA language specification is also a regular AspectLISA specification. In section 3.3 the RobotTime language has been specified as an extension of the Robot language using multiple attribute grammar inheritance. As can be noticed, semantic rule (`COMMAND.time=1;`) has to be repeated in several productions (`COMMAND ::= left`, `COMMAND ::= right`, `COMMAND ::= up`, `COMMAND ::= down`). New semantics in the RobotTime language can be seen as a new aspect which crosscuts the language structure. Therefore, the RobotTime language can be better specified using aspect-oriented attribute grammars. The RobotTime language specifications written in AspectLISA are shown in Fig. 3. Note that four pointcuts have been specified which match all seven productions in the Robot language. For example, pointcut `Begin` matches production `START ::= begin COMMANDS end` and pointcut `SimpleCommand` matches productions `COMMAND ::= left`, `COMMAND ::= right`, `COMMAND ::= up`, and `COMMAND ::= down`. To each pointcut, advice is attached which define the new semantics of matched productions (e.g., semantic for simple command is that each command spent one time slot `C.time = 1;`).

In [20], the RobotSpeed language has been defined as an extension of the RobotTime language. An additional `speed` construct has been added to the language such that the robot can now move with different speed. The RobotSpeed language can be specified purely with aspect-oriented techniques as shown in Fig. 4. Note that all pointcuts from the RobotTime language have been inherited. Only new advice have to be defined with additional semantics about speed of the movement. Hence, new advice extends previous advice that is inherited.

In Fig. 3 and Fig. 4 illustrative examples of AspectLISA are shown. The approach is scalable to larger languages and has been used in re-specifying the AspectCOOL language [3] which is an aspect-oriented extension of COOL (Class Object-Oriented Language)[5]. Typical examples of aspects in language specifications can be additional code generation, different language extensions (e.g., excep-

[5] Our COOL language should not be confused with the early domain-specific aspect-oriented COOL language by Lopes.

```
language RobotTime  extends Robot {

     attributes double *.time;

     pointcut Begin<START, COMMANDS>  *.start : START ::= .. COMMANDS .. ;
     pointcut SimpleCommand<COMMAND>  *.move  : COMMAND  ::= * ;
     pointcut NoCommands<COMMANDS>  *.moves : COMMANDS ::= epsilon ;
     pointcut SeqCommands<COMMANDS[0], COMMAND, COMMANDS[1]>
                              *.moves : COMMANDS ::= COMMAND COMMANDS ;

     advice Init<S,C>  on Begin {
        S.time = C.time;
     }

     advice SetTime<C>  on SimpleCommand {
        C.time=1;
     }

     advice ClearTime<Cs>  on NoCommands {
        Cs.time=0;
     }

     advice SumTime<C0, CM, C1>  on SeqCommands {
        C0.time = CM.time + C1.time;
     }
}
```

Fig. 3. RobotTime Language using AspectLISA

tion handling, aspects, new paradigms), language specification debugging, attribute tracking.

6 Conclusion

In the paper, aspect-oriented attribute grammars has been proposed and formally defined. The concept has been incorporated into AspectLISA, an aspect-oriented compiler generator based on attribute grammars. Aspect-oriented programming is a very promising approach and has been successfully used in tools for language definition and implementation. Some of the known contributions in this field were reviewed, as a motivation for our proposal. LISA already has mechanisms to support inheritance and modularity. These mechanisms support nicely the notion of object-oriented aspects; on the other side, adding aspects will allow to write simpler specifications avoiding, for example, the repetition of semantic rules. The challenge in programming language definition is also to support reusability and extensibility: aspects will reinforce these features. Aspect-oriented features of the AspectLISA tool increase modularity since different concepts of programming language can be designed and implemented separately in different modules. These modules are also more reusable due to inheritance, which is successfully incorporated into our tool.

7 Acknowledgements

We would like to thank Jeff Gray for useful comments on earlier versions and Mitja Lenič for ideas and tips on implementation.

```
language RobotSpeed  extends RobotTime {
 lexicon {
   Commands speed
   Number [0-9]+  }

 attributes int *.inspeed, *.outspeed;

 rule  extends start {
     compute {
     }
 }

 rule speed {
   COMMAND ::= speed #Number  compute {
     COMMAND.time = 0;    // no time is spent for this command
     COMMAND.outspeed = Integer.valueOf(#Number.value()).intValue();
     // this command does not change the position
     COMMAND.outp = COMMAND.inp;
   };
 }

 advice  extends Init<S,C> {
   C.inspeed = 1;  // beginning speed
   S.outspeed = C.outspeed;
 }

 advice SpeedPropagation  extends SumTime<C0, CM, C1> {
    CM.inspeed  = C0.inspeed;     // speed propagation
    C1.inspeed = CM.outspeed;     // to sub-commands
    C0.outspeed = C1.outspeed;
 }

 advice SameTime  extends ClearTime<Cs> {
    Cs.outspeed = Cs.inspeed;
 }

 advice  CalculateTime  extends SetTime<C> {
    C.time = 1.0/C.inspeed;
    C.outspeed = C.inspeed;
 }
}
```

Fig. 4. RobotSpeed Language using AspectLISA

References

[1] ANTLR – ANother Tool for Language Recognition. http://www.antlr.org, 2006.

[2] AspectG. http://www.cis.uab.edu/wuh/ddf/index.html, 2006.

[3] E. Avdičaušević, M. Lenič M. Mernik, and V. Žumer. AspectCOOL: An experiment in design and implementation of aspect-oriented language. *ACM SIGPLAN Notices*, 36(12):84–94, December 2001.

[4] O. de Moor, S. L. Peyton Jones, and E. Van Wyk. Aspect-oriented compilers. In *Generative and Component-Based Software Engineering (GCSE)*, pages 121–133, 1999.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[6] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.

[7] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[8] P. Henriques, M. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu. Automatic generation of language-based tools using LISA. *IEE Proceedings - Software Engineering*, 152(2):54–69, April 2005.

[9] M. Jourdan, D. Parigot, C. Julie, O. Durin, and C. Le Bellec. Design, implementation and evaluation of FNC-2 attribute grammar system. In *Proc. of the ACM Sigplan'90 Conference on Programming Language Design and Implementation*, pages 209–222, 1990.

[10] K. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In N.M.-O. Horatiu Cirstea, editor, *Proceedings of the 6th International Workshop of Rule-Based Programming (RULE)*. ENTCS, Nara, Japan, Elsevier, April 2005.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM (Special issue on Aspect-Oriented Programming),* 44(10):59–65, October 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP97 Object-Oriented Programming, Lecture Notes in Computer Science*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[13] P. Klint, T.van der Storm, and J.J. Vinju. Term rewriting meets aspect-oriented programming. Technical report, Centrum voor Wiskunde en Informatica, 2004.

[14] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[15] J. Kollár and M. Tóth. Temporal logic for pointcut definitions in AOP. *Acta Electrotechnica et Informatica*, 5(2):15 – 22, 2005.

[16] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in aspectC++. In *GPCE*, pages 55–74, 2004.

[17] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 2005. To appear.

[18] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, September 2000.

[19] M. Mernik, X. Wu, and B. Bryant. Object-oriented language specifications: Current status and future trends. In *ECOOP Workshop: Evolution and Reuse of Language Specifications for DSLs (ERLS)*, 2004.

[20] M. Mernik and V. Žumer. Incremental programming language development. *Computer Languages, Systems and Structures*, (31):1–16, 2005.

[21] J. Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196 – 255, 1995.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[23] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[24] C. A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison Wesley, Second edition, 2002.

[25] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The Asf + Sdf meta-environment: A component-based language development environment. *Lecture Notes in Computer Science*, 2027:365–370, 2001.

[26] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1370–1374, New York, NY, USA, 2005. ACM Press.

[27] E. Van Wyk. Aspects as modular language extensions. *Electronic Notes in Theoretical Computer Science*, 82(3), 2003.

The Language Evolver Lever — Tool Demonstration —

Elmar Juergens [1,2] Markus Pizka [1]

Institut für Informatik, Technische Universität München
Boltzmanstr. 3, 85748 Garching, Germany

Abstract

Since many domains are constantly evolving, the associated domain specific languages (DSL) inevitably have to evolve too, to retain their value. But the evolution of a DSL can be very expensive, since existing words of the language (i. e. programs) and tools have to be adapted according to the changes of the DSL itself. In such cases, these costs seriously limit the adoption of DSLs.

This paper presents Lever, a tool for the evolutionary development of DSLs. Lever aims at making evolutionary changes to a DSL much cheaper by automating the adaptation of the DSL parser as well as existing words and providing additional support for the correct adaptation of existing tools (e. g. program generators). This way, Lever simplifies DSL maintenance and paves the ground for bottom-up DSL development.

Key words: domain specific languages, bottom- up language development, language evolution, coupled transformation

1 Introduction

Just as other software artifacts, languages need to evolve as the environments in which they are employed change.

This is especially apparent for domain specific languages (DSLs), since they are usually tightly bound to a domain.

Whenever its domain evolves, a DSL must be adapted in order to reflect these changes. Evolving a DSL requires three main steps:

- evolution of the language syntax
- migration of existing words (i. e. programs) to conform to the new grammar

[1] Email: {juergens, pizka}@in.tum.de
[2] Thanks to Jurgen Vinju for plentiful advice on SDF/SGLR, Ralf Lämmel for valuable discussion on Grammar Adaptation and Jonathan Streit for helpful comments on this paper.

- adaptation of language processing tools (i. e. parser, generator)

In traditional approaches to implement DSLs, the evolution capabilities are limited, since all three evolution steps usually have to be performed manually. Transformers that migrate existing words must be written and parsers and generators must be adapted by hand. Costs for doing this in an ad-hoc manner every time a DSL evolves are high and thus seriously inhibit evolution.

This paper explains our solution to the evolutionary development of DSLs which we call Lever (Language Evolver). Lever provides itself a domain specific language for DSL creation and evolution. It automates the adaptation of a DSL's syntax, parser and existing words. Furthermore, it supports the manual adaptation of the DSL generator by indicating those generator parts that are affected by the language evolution operations performed.

Related Work

A tool having a strong relation to Lever is TransformGen [1,5]. It simplifies the migration of existing words, but provides only limited support for coupled evolution and does not support the adaptation of parsers or generators.

The grammar evolution part of Lever was inspired by the work of Ralf Lämmel on Grammar Adaptation [3,4] but heads into a different direction by considering coupled evolution operations on words and their grammars.

2 Overview of Language Evolution with Lever

2.1 Grammar Evolution

Lever uses labeled context free grammars for the specification of the syntax of a language. A labeled context free grammar extends canonical context free grammars with unique labels for productions and production symbols. These labels are later on used in path expressions that navigate through labeled context free grammars to select grammar elements.

Grammars in Lever are mutable. Lever provides a Grammar Evolution Language that is used to create and modify grammar elements. The Grammar Evolution Language comprises a set of evolution operations that is complete in the sense that every grammar can be turned into any other grammar by applying a sequence of Grammar Evolution Language statements.

2.2 Word Evolution

Lever internally represents words as labeled derivation trees: The production labels name the nodes and the production symbol labels name the edges in the tree. Thus, the same path expressions that select grammar elements from the labeled context free grammar can be used to select corresponding nodes from the labeled derivation tree. This turns path expressions into a uniform querying mechanism for both labeled context free grammars and derivation trees.

Labeled derivation trees are also mutable. Dual to the Grammar Evolution Language, Lever provides a Word Evolution Language that is used to perform evolution operations on the derivation trees. The Word Evolution Language comprises a set of evolution operations that is complete in the sense that every derivation tree can be turned into any other derivation tree by applying a sequence of Word Evolution Language statements.

2.3 Coupled Evolution of Grammar and Words

While the grammar and word evolution languages are expressive, their level of abstraction is still relatively low, since they target grammar and word evolution separately.

Various frequently used evolution operations can be done more comfortably using higher level coupled evolution operations that are automatically mapped onto corresponding grammar and word evolution operations. Examples for such higher level commands are renaming of terminals or the introduction of new nonterminals with a default value. Lever provides an integrated Language Evolution Language to facilitate such coupled evolution operations. It builds on the Grammar- and Word Evolution Languages to implement these coupled evolution commands. When working with Lever, users mainly employ the Language Evolution Language. Only in cases it does not cover, elementary grammar and word evolution operations are used.

The Language Evolution Language is extensible, allowing users to add their own coupled evolution commands. This way we hope to gradually grow it until it provides all commonly encountered language evolution operations.

It is interesting to notice that the Language Evolution Language itself is a DSL that is being developed in a bottom-up, stepwise manner and could thus be implemented using Lever. However, the Language Evolution Language is currently realized as an internal DSL, since the Grammar- and Word Evolution Languages are still evolving, as our understanding of grammar and tree transformations changes. It is planned to implement the Language Evolution Language using Lever, as soon as the Grammar- and Word Evolution Languages reach a sufficient level of stability.

2.4 Adaptation of Language Processing Tools

Lever can automatically produce parsers for its languages. It generates SDF grammars [2] from labeled context free grammars and uses the SGLR parser [6] to instantiate labeled derivation trees from words of the language. Adaptation of the parser is thus completely automated. [3]

[3] Note that Lever does not depend on GLR parsing techniques. If their use is not desired, they can be replaced by hand-written parsers. However, parser adaptation then cannot be

Language processing tools (i. e. generators) use path expressions to access nodes in labeled derivation trees. Lever does not automate the adaptation of these path expressions after language evolution, yet. But path expressions are grammar- aware: Lever validates path expressions statically against the grammar to detect those expressions that would fail or only produce empty result sets when evaluated on labeled derivation trees. This static checking detects all path expressions that broke during language evolution.

3 Demonstration

We demonstrate an exemplary evolution step to illustrate the stepwise development of a simple DSL to generate data structures. The initial grammar is displayed textually[4] and visually[5] in Figures 1 and 2a. It contains two parameters that influence code generation: The *type* describes allowed data objects and when *unique* is present, there may be no two equal objects contained in store instances. Figures 3, and 2b show words for the initial grammar (both textually and visually).[6]

```
"Store" "[" lbl:"type=" type:"[a-z]+" "unique"? "]" -> Datastructure {Store}
```

Fig. 1. Initial Grammar in textual form

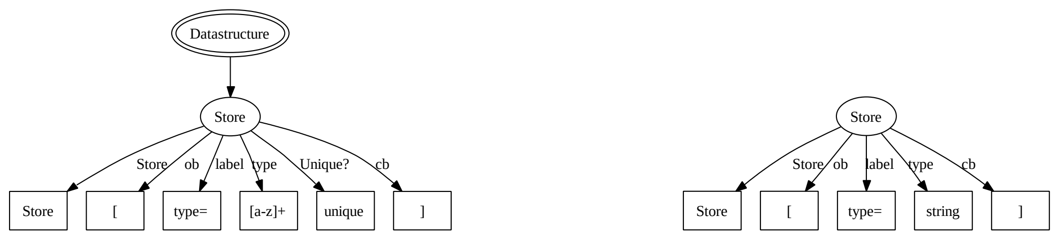


Fig. 2. a) Initial Grammar b) Initial Word

```
Store [type=string]                        Bag [type=string]
Store [type=object unique]                 Set [type=object]
```

Fig. 3. Words before and after evolution

As our understanding of the domain of data structures grows, we decide to replace the *unique* keyword with the terms *Set* and *Bag*. To reflect our changed understanding of the domain, we evolve our DSL accordingly:

- All unique Stores are to be converted to Sets, all other instances to Bags.
- Both data structures contain the type parameter. To avoid duplication in the resulting grammar, we encapsulate it into a production of its own.

automated anymore.

[4] In this example, some of the labels of literal symbols have been omitted for brevity.

[5] Key: double ellipses are sorts, single ellipses are productions and boxes are terminals.

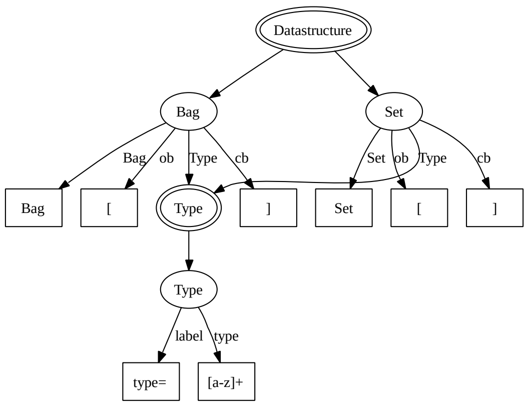
[6] Key: ellipses are nodes corresponding to productions, boxes are leafs with word fragments.

- The Store production is now unused and gets removed from the grammar.

Figures 3, 4, 5a and 5b display grammar and words after evolution.

```
"Set" ob:"[" Type cb:"]" -> Datastructure {Set}
"Bag" ob:"[" Type cb:"]" -> Datastructure {Bag}
lbl:"type=" type:"[a-z]+" -> Type {Type}
```

Fig. 4. Textual representation of the evolved grammar



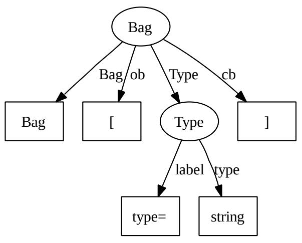


Fig. 5. a) Evolved grammar b) Evolved word

Figure 6 shows the Language Evolution Language commands for these evolution operations.[7]

```
encapsulate "lbl", "type" into Type in Store

create production "Set" ob:"[" Type cb"]"       ->      Datastructure {Set}
create production "Bag" ob:"[" Type cb"]"       ->      Datastructure {Bag}

for store in Stores:
       remove literal "Store"
       if contains literal unique:
             set production to "Datastructures.Set"
             append leaf "Set"
       else:
             set production to "Datastructures.Bag"
             append leaf "Bag"

delete production "Store"
```

Fig. 6. Language Evolution Language statements

[7] The syntax of the statements has been simplified to increase readability.

4 Conclusion

Lever provides several DSLs for different levels of language evolution: the Grammar Evolution Language for grammars, the Word Evolution Language for words (i. e. programs) and the Language Evolution Language for the coupled evolution of grammar and words. Evolution operations formulated using these DSLs allow Lever to automate the adaptation of existing words and parsers. Furthermore, Lever can point out areas that need manual adaptation in tools that do not get adapted automatically (e. g. generators). Compared to ad hoc approaches to DSL evolution, Lever thus significantly decreases evolution costs.

Future work includes the application of Lever to the development of real world DSLs to grow the Language Evolution Language and thus increase its expressiveness. Additionally, we plan to automatically adapt path expressions for those Language Evolution Language commands that merely refactor a language (i. e. renaming of nonterminals, encapsulating or inlining nonterminals,...).

Lever is currently being implemented and tested and will be made available in the first half of 2006.

References

[1] Garlan, D., C. W. Krueger and B. S. Lerner, *Transformgen: automating the maintenance of structure-oriented environments*, ACM Trans. Program. Lang. Syst. **16** (1994), pp. 727–774.

[2] Heering, J., P. R. H. Hendriks, P. Klint and J. Rekers, *The syntax definition formalism sdf reference manual*, SIGPLAN Not. **24** (1989), pp. 43–75.

[3] Lämmel, R., *Grammar Adaptation*, in: *Proc. Formal Methods Europe (FME) 2001*, LNCS **2021** (2001), pp. 550–570.

[4] Lämmel, R. and G. Wachsmuth, *Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment*, in: M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, ENTCS **44** (2001).

[5] Staudt, B. J., C. W. Krueger and D. Garlan, *A structural approach to the maintenance of structure-oriented environments*, in: *SDE 2: Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments* (1987), pp. 160–170.

[6] van den Brand, M. G. J., J. Scheerder, J. J. Vinju and E. Visser, *Disambiguation filters for scannerless generalized lr parsers*, in: *CC '02: Proceedings of the 11th International Conference on Compiler Construction* (2002), pp. 143–158.

Combining Deep and Shallow Embeddings

Joni Helin [1,2]

Institute of Software Systems
Tampere University of Technology
Tampere, Finland

Abstract

This paper presents an approach for verifying translator correctness when the source language has formal semantics. Instead of verifying the translator implementation, a novel language mechanization combination is devised to reduce total complexity involved. A deep embedding is defined to serve as a baseline for specification meaning. For each specification, an equivalence proof is constructed and conducted to ensure that the translated shallow representation is semantically equivalent to the deep representation. Structure of an equivalence proof is systematic and can be derived from specification structure mechanically. The use of two embeddings also affects the embeddings favourably by enabling them to be defined in a simpler manner.

Key words: formal specification, translator correctness, shallow embedding, deep embedding, Ocsid

1 Introduction

Language implementation is a large part of the effort whether we are designing a programming language or a formal language. If ultimate confidence in tool correctness is to be attained, it usually requires verifying that the implementation has the desired properties that constitute correctness. However, reasoning about programming language constructs is difficult and a translator tends to be a fairly complex piece of code. Therefore we would prefer another approach where the same results are achieved by a smaller total effort.

In this paper we propose an alternative approach to tool correctness when the source and target languages have formal semantics. Instead of verifying the translator implementation, we exploit two simultaneous translations. The goal of the approach is to reduce the total amount of effort and complexity involved by taking smaller steps.

[1] Joni Helin was funded by Academy of Finland grant 201064.

[2] Email: joni.helin@tut.fi Fax: +358 3 3115 2913

The approach entails the construction of a semantic yardstick to which the results of the original, shallow, translation can be compared. For this we use a deep embedding, which is essentially a formal language definition. A deep embedding also allows for added confidence in the meaningfulness of our language definition; we can state properties that it should exhibit and use a theorem prover to verify them.

The use of two translations suggests that we incur some extra work, but we argue that in the right circumstances, the required effort is an order of magnitude smaller. The key factor in the reduced complexity lies in the structure of equivalence proofs; they can be systematically derived from specifications.

For the applicability of the approach presented in this paper, the exact details of the embedded language is not significant. What is necessary is to have a precise meaning associated with the programs or specifications written in the language. We frame the discussion of our approach with the mechanization of an experimental specification language Ocsid to a host logic provided by the theorem prover PVS.

The rest of the paper is organized as follows. Section 2 discusses approaches to mechanization of programming logics. Section 3 presents the specification language Ocsid and in Section 4 it is employed to demonstrate a concrete application of the technique advocated. Section 5 discusses issues related to producing and conducting equivalence proofs. Section 6 takes a look at related approaches and Section 7 concludes the paper.

2 Approaches to Embedding Programming Logics

As we are developing a formal specification language, we naturally have to provide users with the ability to formally reason about properties of their specifications. Instead of building a dedicated proof system, reasoning support is often provided by mechanizing the language in a higher order logic, i.e. embedding the language inside a general theorem prover logic [5]. There are two main embedding schemes, *shallow* and *deep*. For the purposes of this paper, we only give a brief explanation of their differences here. The reader is referred to a more thorough discussion offered in [2].

In a shallow embedding the meaning of a program or specification is retained by a translation into a semantically equivalent representation in a host logic. In comparison, a deep embedding explicitly represents the grammar of a language in the host logic and provides semantic functions for interpreting their meaning. This gives the opportunity to reason about the language itself, not just concrete programs.

It is easy to see that the less indirection a language embedding entails, the more straightforward it is to reason about embedded programs. A deep embedding is also challenging for automated decision procedures of theorem provers, which are designed for handwritten specifications. For these reasons shallow embeddings are often preferable in actual verification.

From the point of view of language designers, it is desirable to gain confidence in the soundness of the language. While a shallow embedding is easy to work with, it does not facilitate full reasoning about language properties or classes of programs. This is because syntactic constructs are not retained as first class values, but are only implicit in the structure of the embedded program. The link between a language definition and the shallow translation is informal but usually highly non-trivial. Thus confidence in implementation correctness is necessarily lacking. As a deep embedding does not suffer from these limitations, it may sometimes be justified to sacrifice the benefits of a shallow embedding.

As is evident from the above discussion, both embedding approaches have their respective strengths and weaknesses. Instead of committing to one or the other, this paper goes on to show how they can be beneficially used together.

3 Ocsid Overview

In order to fix the discussion into a concrete setting, we use the experimental specification language Ocsid. To better illustrate the approach advocated in the paper, we only present a brief introduction to the underlying ideas and simplify the technical details. The interested reader is referred to [6] for a more complete discussion of full features and methodological implications.

Ocsid is a specification language for *view-based* decomposition of distributed collaboration. It is a state-oriented formalism whose semantics are based on the Temporal Logic of Actions (TLA) [7]. Ocsid employs an object-based framework where data is modeled using classes, objects, and data members. A system consists of an arbitrary number of objects. Operations on data members are expressed using *joint actions* [3], which model synchronizations of participating objects in a collaboration.

A joint action consists of a role list which declares formal participants, a guard and a body. The guard determines a joint action's eligibility for execution for a particular combination of participating objects. The body of a joint action specifies the collective effects on data members of participants, but leaves out the communication protocol necessary for achieving them. Execution is nondeterministic in that in each state of an execution, one of the enabled actions becomes selected. Concurrency is modelled by interleaving, i.e. independent actions can occur in any order.

The view mechanism of Ocsid allows the modularization of specifications to match behavioral concerns. Each view only specifies the aspects of structure and behavior needed in the verification of a particular *view invariant*, and the composition of views is defined to preserve the invariants of the views. To make this feasible, views are closed. In closed-world modeling, if a specification contains a state variable, it also contains all operations that assign to it. This *mutational completeness* allows for reasoning about temporal safety properties by considering views in isolation.

A contrived example view is given below. It states that when the value of variable x is equal for two participating objects, the value of the first object is incremented. A system satisfies this view if at each step of execution it either stutters (leaves variables unchanged) or two eligible objects of class C syncronize to update variable x as the action A specifies.

```
view simple is
  class C;
  variable C.x: integer;

  action A by c1, c2: C
  when c1.x = c2.x do
    c1.x := c1.x + 1;
  end
end
```

4 Embedding Ocsid

This section demonstrated the use of two simultaneuous embeddings for mechanizing the Ocsid language. As a host logic we employ that of PVS [9], which is suitable because of its higher order nature. Another advantage of PVS is that its logic contains an executable subset [10] which can be used as a functional programming language. A ground evaluator is provided for evaluating executable PVS functions. This allows for computing the results of syntactic transformations.

Figure 1 shows a depiction of how the basic elements of the mechanization arrangement are related.

4.1 Common Semantic Base

PVS type declarations given below define the semantic base for the temporal elements of Ocsid. These are shared among the shallow and deep embeddings. This is important because it facilitates later connecting the two embeddings.

```
STATE: TYPE+
BEHAVIOR : TYPE = sequence[STATE]
STEP : TYPE = [STATE, STATE]
ACTION : TYPE = [STEP -> bool]
SPECIFICATION : TYPE = pred[BEHAVIOR]

class[ref_type: TYPE+]: DATATYPE
BEGIN
  object(reference: ref_type): object?
END class
```

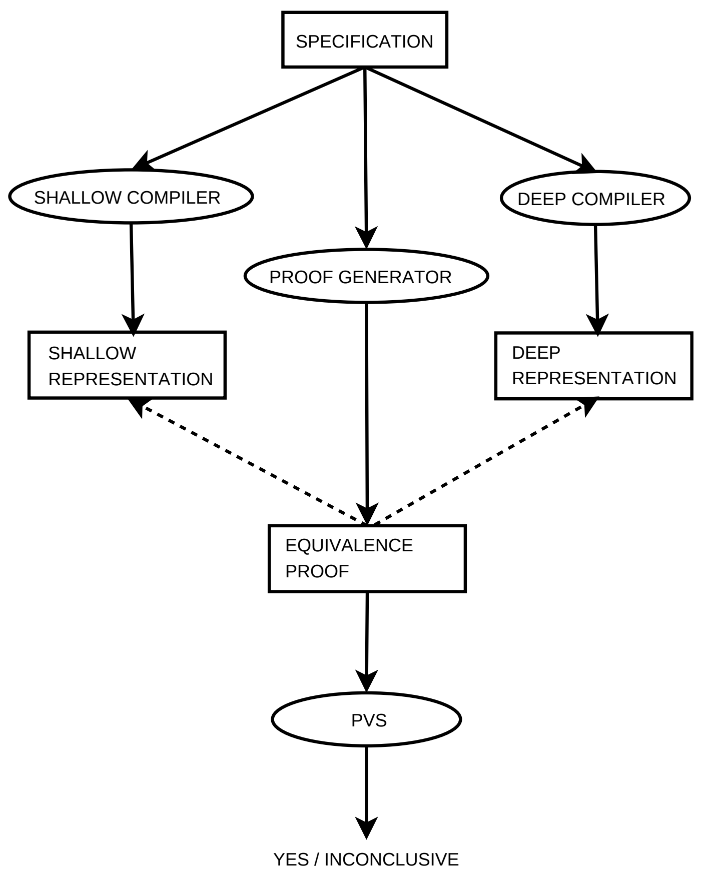


Fig. 1. Elements of Mechanization

Here STATE is an uninterpreted type which defines the set of all possible states across all possible specifications and behaviour is simply a sequence of states. An action is a predicate on steps (pairs of states), i.e. it decides whether the action is enabled in the starting state and is the specified result state possible for the participating combination. Similarly, a specification is semantically a predicate on behaviours.

At this point we do not provide definitions for our data facilities, i.e. variables and values. They are required when we go into details about actions, but since their representation is dependend on the embedding used, they can not be included in the common base.

4.2 Deep Embedding of Ocsid

Shown below is the abstract syntax of Ocsid views using algebraic data types of PVS. This description is very similar to the traditional use of BNF for the same purpose. Each data type definition enumerates constructors for making values (sentences), lists their parameters and provides recognizer predicates for determining applied constructors. Parameters implicitly define accessor functions to constructor arguments.

A subset of syntactic Ocsid expressions is shown when their semantics are defined.

The use of a deep embedding allows us to formulate and prove properties of syntactic operations. For example, proving composition in Ocsid simply entails syntactic induction over all possible pairs of views. However, Ocsid-specific syntactic transformations are omitted as they are not relevant to the embedding approach advocated in this paper.

```
type_syntax : TYPE = nameid

class_syntax : DATATYPE
BEGIN
  class(name : nameid) : class?
END class_syntax

variable_syntax: DATATYPE
BEGIN
  variable(name: nameid, class: class_syntax,
           type: type_syntax): variable?
END variable_syntax

formal_parameter_syntax: DATATYPE
BEGIN
  formal_param(name: nameid, type: type_syntax): formal_param?
  formal_role(name: nameid, class: class_syntax): formal_role?
END formal_parameter_syntax

assignment_syntax: DATATYPE
BEGIN
  assignment(obj: (formal_role?), field: variable_syntax,
             rhs: expr_syntax): assignment?
END assignment_syntax

action_syntax: DATATYPE
BEGIN
  action(name : nameid, roles : list[formal_parameter_syntax],
         guard : list[expr_syntax],
         body : list[assignment_syntax]): action?
END action_syntax

view_syntax: DATATYPE
BEGIN
  view(name: nameid,
       classes: list[class_syntax],
       variables: list[variable_syntax],
       actions: list[action_syntax]): view?
END view_syntax
```

In this demonstration we define a bare-bones embedding from which typing information is intentionally left out. This is justifiable because we have two embeddings where the deep one is intended mainly as a yardstick for specification behaviour. Thus, classic static analysis such as type correctness can be fairly confidently assumed to be handled by the PVS type system in the shallow embedding. This greatly reduces the size and complexity of the deep embedding. A constraint that comes with this simplification is that it is no longer possible to define syntactic transformations which depend on typing information, but those are not needed here.

The semantic domain of the deep embedding is given below. First, the uninterpreted type OBJECT declares the sets of all objects. Algebraic data type VALUE provides constructors, accessors and recorgnizers for values in our simplified typing scheme. VARIABLE is defined as a mapping from an object and a state to a value. Uninterpreted mapping functions represents connections from syntactic representations of classes, types and variables to their semantic counterparts. Environment is a record which binds action roles and other parameters in the syntactic domain to the semantic domain.

```
  OBJECT: TYPE
  VALUE: DATATYPE
  BEGIN
    error_value: error_value?
    integer_value(intval: integer): integer_value?
    boolean_value(boolval: boolean): boolean_value?
    reference_value(refval: object): reference_value?
    lambda_value(lambdaval: [value -> value]): lambda_value?
  END VALUE

  VARIABLE: TYPE = [OBJECT, STATE -> VALUE]

  type_map: [nameid -> pred[VALUE]],
  class_map: [class_syntax -> pred[OBJECT]],
  var_map: [variable_syntax -> VARIABLE],

  environment: TYPE = [# objmap: [(formal_role?) -> OBJECT],
                         parmap: [(formal_param?) -> VALUE] #]
```

The more interesting semantic functions are given next. View semantics are defined as follows. It states that for a behaviour to satisfy the given view specification, for all steps in the behaviour there has to be an applicable action or the step must be stuttering (values of the variables of the view do not change).

```
sem(v: view_syntax): SPECIFICATION =
  LAMBDA (b: BEHAVIOR):
    (FORALL (n: nat):
      (EXISTS (a: action_syntax):
         member(a, actions(v)) AND
            sem(a, classes(v), variables(v))(b(n), b(n + 1)))
      OR sem(stuttering_action, classes(v), variables(v))
            (b(n), b(n + 1)))
```

Action semantics go into details about the guard and body. The first function is a simple forwarding to a recursive function that performs the actual processing. First, the else branch binds suitable existentially quantified participant objects to roles. The then part first ensures that all guard conjuncts are satisfied in the first (unprimed) state. Second, it iterates over all classes of the view and by universally quantifying over their objects it ensures that the values of all their fields in the second (primed) state correspond to the value assigned to them in the body.

```
sem(a: action_syntax,
    classes: list[class_syntax], vars: list[variable_syntax]):
ACTION =
  LAMBDA (now, next: STATE):
    sem(a, now, next, empty_env, roles(a), classes, vars)

sem(a: action_syntax, unprimed, primed: STATE,
    env: environment, roles: list[(formal_role?)],
    classes: list[class_syntax], vars: list[variable_syntax]):
RECURSIVE bool =
  IF null?(roles) THEN
    every((LAMBDA (g: expr_syntax):
            boolval(sem(g, env, unprimed))), guard(a))
    AND
      every((LAMBDA (class: class_syntax):
        (FORALL (obj: (class_map(class))):
          (every ((LAMBDA (field : variable_syntax):
            (var_map(field)(obj, primed)
              = primed_value(body(a), unprimed, env)
                  (obj, field))),
                 filter((LAMBDA (f: variable_syntax):
                   class(f) = class), vars))))),
            classes)
  ELSE
    EXISTS (obj: (class_map(class(car(roles))))):
      sem(a, unprimed, primed,
          env WITH [‘objmap := objmap(env)
                      WITH [(car(roles)) := obj]],
          cdr(roles), classes, vars)
  ENDIF
MEASURE length(roles)
```

Function primed_value is responsible for calculating the right hand sides of assignments in an action body. It constructs a mapping from objects and fields to their primed values, which default to stuttering if no explicit assignment is specified for a combination.

```
primed_value(assignments: list[assignment_syntax],
             unprimed: STATE, env: environment):
RECURSIVE [OBJECT, variable_syntax -> VALUE] =
  IF null?(assignments)
    THEN LAMBDA (obj: OBJECT, field: variable_syntax):
      var_map(field)(obj, unprimed)
  ELSE CASES car(assignments)
         OF assignment(role, f, rhs):
              primed_value(cdr(assignments), unprimed, env)
                WITH [(objmap(env)(role), f)
                         := sem(rhs, env, unprimed)]
         ENDCASES
  ENDIF
   MEASURE length(assignments)
```

Selected syntactic expressions and corresponding semantic functions are given below. They are fairly straightforward mappings betweeen syntactic and semantic domains. Universal quantification and the lambda form are included to demonstrate the use of object and type predicates and environment manipulation.

```
expr_syntax : DATATYPE
BEGIN
   intconst(v : int): intconst?
   var_ref(obj : (formal_role?), field: variable_syntax): var_ref?
   plus(e1, e2: expr_syntax): plus?
   equal(e1, e2 : expr_syntax): equal?
   forall_q(q_var : (formal_role?), e : expr_syntax): forall_q?
   lambda(param: (formal_param?), body: expr_syntax): lambda?
   application(comp, arg: expr_syntax): application?
END expr_syntax

sem(e: expr_syntax, env : environment, s : STATE):
RECURSIVE VALUE =
  CASES e
    OF intconst(i): integer_value(i),
       var_ref(role, f): var_map(f)(objmap(env)(role), s),
       plus(e1, e2): integer_value(  intval(sem(e1, env, s))
                                   + intval(sem(e2, env, s))),
       equal(e1, e2): boolean_value(sem(e1, env, s)
                                        = sem(e2, env, s)),
       forall_q(qvar, e):
         boolean_value(FORALL (o: (class_map(class(qvar)))):
           boolval(sem(e, env WITH [‘objmap := objmap(env)
                                       WITH [(qvar) := o]], s))),
       lambda(pr, body): lambda_value(LAMBDA (v: VALUE):
         IF type_map(type(pr))(v) THEN
           sem(body, env WITH [‘parmap := parmap(env)
                                  WITH [(pr) := v]], s)
          ELSE error_value ENDIF),
       application(comp, arg): lambdaval(sem(comp, env, s))
                                 (sem(arg, env, s))
    ENDCASES
   MEASURE e BY <<
```

4.3 Shallow Embedding

Shallow embedding is implemented by the Ocsid compiler. As Ocsid is carefully defined to employ the type and expression subset of the PVS logic, the mapping into corresponding PVS declarations and definitions is a fairly straightforward. As the exact translation schema is not important for the approach, it is not rigorously specified here. Relevant portions of the mapping of the example view simple given earlier are shown below.

```
simple: THEORY
BEGIN
  C_ref : TYPE+
  C : TYPE = class[C_ref]
  x : TYPE = [[C, state] -> integer]

  A : ACTION = LAMBDA (st : STEP) :
    EXISTS (  c1: C , c2: C ):
      guard( x(c1 , now(st)) = x(c2 , now(st)) ) AND
      body( (FORALL ( C_o : C ):
        IF C_o = c1 THEN
          x(c1, next(st)) = x(c1 , now(st)) + 1
        ELSIF C_o = c2 THEN
          x(c2, next(st)) = x(c2 , now(st))
        ELSE
          x(C_o, next(st)) = x(C_o, now(st))
        ENDIF) )

  SPEC : SPECIFICATION =
    LAMBDA (b : BEHAVIOR): FORALL (n : nat) :
      LET st = (b(n), b(n+1)) IN A(st) OR stutter(st)
END simple
```

It should be evident that this form of embedding is desirable from the point of view of reasoning about specifications. Each action directly corresponds to a similarly named function which existentially quantifies over the role declarations. The actual semantics are encoded inside the identity functions guard and body. Because views are closed, they take care to specify the effects of the action for all objects of all classes. The delicate part of the shallow embedding is in the handling of aliasing in the action bodies regarding bound roles.

4.4 Connecting the Embeddings

As we have already described both embeddings, we now have to connect their semantic domains to be able to check specification equivalence. This step would not be possible unless they shared a common semantic base, on which both embeddings are built. In this case the embeddings have been constructed to share the uninterpreted type STATE (and BEHAVIOUR obviously).

The point where they crucially differ is how structure and data is represented, i.e. objects, values and variables. Whereas the shallow embedding maps these specification-wise into corresponding PVS entities, the deep embedding has to manage them at the language level, where it can be seen manifested in the explicit types OBJECT, VALUE and VARIABLE. It is these two worlds that need to be connected for comparing the different representation of an Ocsid specification.

PVS logic offers a theory parametrization mechanism similar to templates or generics in conventional programming languages. We use this parametrization facility to link semantic representations in the deep embedding to their counterparts in the shallow embedding. This is achieved in part by moving appropriate declarations in the body of the theory containing semantic base of the deep embedding to its theory parameter list. These changes are shown below.

```
view_semantics[OBJECT: TYPE,
       var_map: [variable_syntax -> [OBJECT, STATE -> VALUE]],
       type_map:  [type_syntax -> pred[VALUE]],
       class_map: [class_syntax -> pred[OBJECT]]
]: THEORY
BEGIN
...
END
```

As an aside, there are certainly options regarding the desired nature of the connection. In this case it can be seen in the details regarding the different handling of objects and values. We have chosen to bump up the datatype VALUE to the shared semantic base of the two embeddings but to make the OBJECT type specification-specific. This makes types structurally equivalent while classes are by definition mutually disjoint. To support name equivalence without changing the deep embedding, we would need to make the datatype

VALUE specification-specific also. However, this kind of leeway only exists because we have deliberately left typing out of the deep embedding.

To complete linking the embeddings we need to provide arguments for instantiating a concrete theory from the generic one. For this we need to define a suitable wrapping of the shallow entities and use these proxies as theory arguments. This entails producing a type definition to wrap classes of the specification. Also needed is the generation of functions that wrap and map syntactic variables, types and classes to their semantic equivalents. This is shown below for our example view simple given earlier. The importing clause instantiates the view_semantics theory with the generated entities.

```
object: DATATYPE
BEGIN
  c(cobj: C): c?
END object

var_map(v: variable_syntax): variable =
  IF name(class(v)) = "C" THEN
    IF v = x_stx THEN
      lambda (o: object, st: STATE):
         IF c?(o) THEN integer_value(x(cobj(o), st))
         ELSE error_value ENDIF
    ELSE lambda (o: object, st: STATE): error_value ENDIF
  ELSE LAMBDA (o: object, s: STATE): error_value
  ENDIF

type_map(name: nameid): pred[value] =
  IF name = "integer" THEN integer_value?
  ELSIF name = "boolean" THEN boolean_value?
  ELSIF name = "C_ref" THEN
    (LAMBDA (vv: value): reference_value?(vv) AND c?(ref_val(vv)))
  ELSE (LAMBDA (vv: value): false)
  ENDIF

class_map(class: class_syntax): pred[object] =
  IF      class = c_stx THEN c?
  ELSE lambda (o: object): false
  ENDIF

importing view_semantics[object, var_map, type_map, class_map]
```

Using theory parametrization, we retain the decoupling between the two independent embeddings. The only change required is in the deep embedding where we need to make the relevant entities parameteric. It is the responsibility of the translation tool to produce this link theory.

5 Proving Equivalence of Specification Representations

We are now ready to mechanically make the case that the shallow and deep representations of a specification are equal. For this step there are two possibilities. You can either use a theorem prover for manually constructing the equivalence proof or implement a proof generator which exploits knowledge of the specification structure. We first discuss these issues in general and then provide highlights from an equivalence proof for our example specification.

5.1 Informal basis and motivation

An equivalence proof is highly systematic because both embeddings reflect the same meaning, only the representation is different. This facilitates the implementation of a proof generator, which spares users of the tedious handling of minute technical details of the language. Of course, such a tool is relative complex and highly dependant on the proof system upon which the embeddings are defined. But assuming a sufficient user base this would be preferable as the effort expended in constructing the tool would be amortized across all equivalence proofs ever conducted.

An important factor in simplifying equivalence proofs is to maintain structural similarity between the two representations of a specification. This comes into play when we consider equivalence across syntactic transformations. In the shallow case, the compiler performs the transformation and embeds the resulting specification. As long as we implement these transformations in the deep embedding using only the executable subset of the theorem prover logic, we can use a ground evaluator to churn out the final deep embedded version. If the tools are correct, structures of the two embeddings are similar.

It is important to notice that in the automated case we do not end up with new proof obligations regarding the proof generator. If a theorem prover can successfully execute a proof script to conclusion, we do not need to consider the origin of the script, but only trust the theorem prover. So the situation does not change whether we produce the proof by hand or not.

If a proof generator always fails in the sense that it produces erraneous proofs, it is of no value. However, it does have value if it proceduces working proofs in common cases. This property is suitable for incremental development of languages and tools. In the first phase, users can construct equivalence proofs by hand. Then, a crude proof generator is written that can handle the simplest and most common language constructs. When the project reaches reasonable stability and demand, final effort can be expended to fill in support for the more esoteric constructs.

Finally, if we were to compare the required expertise between implementing a proof generator and formally verifying a shallow translator, we would see that there is a significant difference. The former requires conventional software engineering skills whereas the latter requires considerable mathematical ability to formally handle intricate programming language constructs. We strongly feel that verification would hardly be feasible in most cases whereas the advocated use of two embeddings and proof generator could plausibly be.

5.2 Example equivalence proof

To make matters more concrete, we have opted to present selected portions of the equivalence proof for the example specification shown earlier. Due to the amount of minute technical details tackled by an equivalence proof, it would be infeasible and unproductive to show the complete proof tree for even a simple specification. Fortunately, the systematic nature of these proofs allow selected highlights to communicate the flavor of involved reasoning.

PVS proof system is based on sequent calculus, which entails transforming a sequent according to a set of proof rules until a conclusion is reached or failure is evident. For our equivalence proof, at the root of the proof tree we have the sequent shallow_deep_equivalence. As can be expected, it states that for all behaviours, the meanings of both representations must imply each other. The proof starts out with skolemization of behaviour and splitting the proof into subtrees for the two directions of implication. The proof shown has been manually constructed as a proof generator is still subject for future development.

```
shallow_deep_equivalence

|-------
[1]   FORALL (b: behavior): sem(simple_view)(b) <=> simple.SPEC(b)
```

In the first branch, after skolemization, instantiation and simplification, we have sequent 1.2. It shows more detail into the specification structure by enumerating joint actions. Essentially, given that some deep action is satisfied, there has to be a satisfied shallow action for the same step. Of course, in these proofs it is always the shallow version of the corresponding action.

```
shallow_deep_equivalence.1.2

{-1}  (EXISTS (a: action_syntax):
          member(a, (: a_stx :)) AND
           sem(a, (: c_stx :), (: x_stx :))(b!1(n!1), b!1(1 + n!1)))
        OR
        sem(the_stuttering_action, (: c_stx :), (: x_stx :))
           (b!1(n!1), b!1(1 + n!1))
  |-------
[1]   A(b!1(n!1), b!1(1 + n!1))
[2]   stutter(b!1(n!1), b!1(1 + n!1))
```

The next two subgoals are reached after first splitting action-wise and then performing expansion of definitions, skolemization, instantiation and rewriting. All these steps are intellectually undemanding and necessitate only straightforward structure-based bookkeeping of entity relationships between deep and shallow domains. They are thus easily subjected to mechanization.

Sequent 1.2.1.1.1 is uncomplicated, stating implication for the guard of action A. The expression level is now visible and displays why the shallow representation is preferable for reasoning purposes. The explicit bindings carried in the deep representation require recurring expanding and make proofs profuse (*an_env* is the empty environment). Establishing this goal is a simple matter of rewriting and propositional simplification.

```
shallow_deep_equivalence.1.2.1.1.1 :

[-1]  v(sem(equal(var_ref(c1_role, x_stx),
                  var_ref(c2_role, x_stx)),
            an_env WITH [(c1_role) := obj!1,
                         (c2_role) := obj!2], b!1(n!1)))
  |-------
[1]   guard(x(obj!1, now(b!1(n!1), b!1(1 + n!1))) =
             x(obj!2, now(b!1(n!1), b!1(1 + n!1))))
```

Sequent 1.2.1.1.2 states implication for the body of action A. Figure 2 shows a graph for a proof subtree reaching a positive conclusion. The proof proceeds by managing aliasing via splitting and purposeful instantiation, followed by standard reasoning. The idea of the graph is not communicate individual steps but to give some intuition to the details involved.

```
shallow_deep_equivalence.1.2.1.1.2 :

[-1]  FORALL (obj: (class_map(c_stx))):
        (var_map(c_stx, x_stx)(obj, b!1(1 + n!1)) =
          primed_value((: assignment(c1_role,
                                     x_stx,
                                     plus
                                     (var_ref(c1_role, x_stx),
                                      intconst(1))) :),
                       b!1(n!1),
                       an_env WITH [(c1_role) := obj!1,
                                    (c2_role) := obj!2])
                      (obj, x_stx))
  |-------
```

```
[1]    IF C_o!1 = obj!1
         THEN x(obj!1, next(b!1(n!1), b!1(1 + n!1))) =
               1 + x(obj!1, now(b!1(n!1), b!1(1 + n!1)))
       ELSIF C_o!1 = obj!2
         THEN x(obj!2, next(b!1(n!1), b!1(1 + n!1))) =
               x(obj!2, now(b!1(n!1), b!1(1 + n!1)))
       ELSE x(C_o!1, next(b!1(n!1), b!1(1 + n!1))) =
             x(C_o!1, now(b!1(n!1), b!1(1 + n!1)))
       ENDIF
```

Sequent 2.1.2.1.2.1.2.2.1.2 shows the other direction of implication for the body of action A, at a slightly more expanded level. Proving this is similar in associated complexity, but differs due to variations in the embedding schemes. For example, there is no instantiation related to branching on role participants as the semantics of the deep embedding produce variable values on demand.

```
shallow_deep_equivalence.2.1.2.1.2.1.2.2.1.2 :

[-1]   IF obj!1 = c1!1
         THEN x(c1!1, next(b!1(n!1), b!1(1 + n!1))) =
               1 + x(c1!1, now(b!1(n!1), b!1(1 + n!1)))
       ELSIF obj!1 = c2!1
         THEN x(c2!1, next(b!1(n!1), b!1(1 + n!1))) =
               x(c2!1, now(b!1(n!1), b!1(1 + n!1)))
       ELSE x(obj!1, next(b!1(n!1), b!1(1 + n!1))) =
             x(obj!1, now(b!1(n!1), b!1(1 + n!1)))
       ENDIF
  |-------
{1}    IF (obj!1, x_stx) = (c1!1, x_stx)
         THEN (var_map(c_stx, x_stx)(obj!1, b!1(1 + n!1)) =
                sem(plus(var_ref(c1_role, x_stx), intconst(1)),
                    an_env WITH [(c1_role) := c1!1,
                                 (c2_role) := c2!1],
                    b!1(n!1)))
       ELSE (var_map(c_stx, x_stx)(obj!1, b!1(1 + n!1)) =
              var_map(class(x_stx), x_stx)(obj!1, b!1(n!1)))
       ENDIF
```

We have now gone over the more representative portions of the equivalence proof with the intention to suggest that no only are these proofs feasible to produce by hand, they are amenable to automated generation. As it comes to omissions, we left out the handling of stuttering as it lacks characteristics not found in other actions. We also skipped details about type conditions

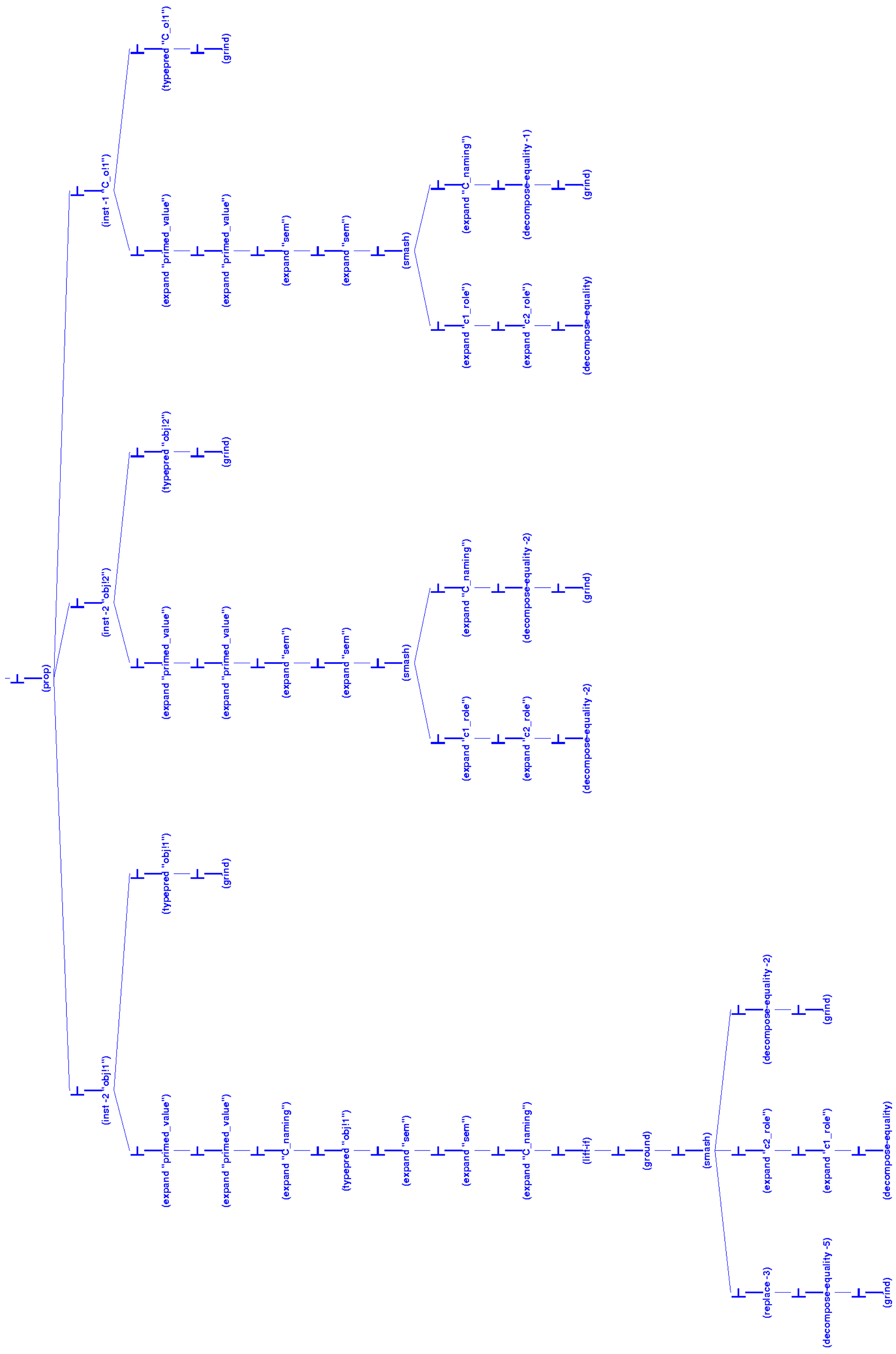


Fig. 2. Proof subtree for sequent .1.2.1.1.2

related to instantiation, as these are handled simply by making sure decision procedures have suitable type predicates to apply.

If we consider proof generation, it is fairly evident that the complexity lies in accurately predicting the results of applied proof steps. For example, when splitting, we need to be able to predetermine what resulting sequent corresponds to which specification-level subgoal to continue proof generation. Fortunately, if this could not be overcome statically, PVS provides support for defining custom proof strategies. They have the ability dynamically determine sequent characteristics and act accordingly. Thus we could define proof strategies for handling each specification-language specific proof subgoal kind and have them invoke each other. While generic, they would have to parametrized to receive relevant information about the specification at hand.

6 Related Work

Structural embeddings are introduced as orthogonal to shallow and deep embeddings in [8]. While they do not attempt to directly tackle the translation credibility problem addressed in this paper, their work has consequences to that issue. They advocate the use of an embedding approach where instead of defining proprietary expression languages, designers could simply borrow those parts from general host logics. They argue that often the exact details of expressions and typing are not central to higher level constructs. These are usually the motivation for new language development because they provide support for the important methodological aspects. Borrowing parts from the underlying logic greatly reduce complexity of an embedding.

Hybrid embedding is introduced in [1]. It is similar to a very shallow translation in that it takes a fairly pragmatic view to mechanical verification. Aspects of complex languages which are problematic to embed are simply not embedded. Instead, they are directly implemented in a programming language, perhaps with the very one underlying the targeted proof system for connectivity purposes.

In [4] a shallow embedding is extended to facilitate features associated with deep embeddings. Because shallow embeddings lack syntactic representation, they do not support syntactic reasoning. They come close to achieving the same effect with their *implicit syntax* approach to formal metatheory. To handle syntactic information, they define inductive predicates over semantic domains. The idea is to express *representability* of a construct. However, while they mostly do away with the need for a deep embedding, the shallow embedding is further complicated.

In [11], the authors introduce a methodology similar to that employed in this paper. For ensuring compiler correctness they use the *translation validation* approach. Rather than proving the correctness of the entire compiler, the correctness of each individual compilation is proved instead. The compiler also doubles as a proof generator by outputting a proof script that can be mechanically checked. Thus their approach to ensuring correctness shares many characteristics discussed in this paper, but the use of two simultaneous embeddings brings additional benefits which are discussed in the next section.

7 Conclusions

In this paper we have suggested an alternative approach to translator correctness when the target language is formal. Instead of verifying a programming logic implementation against the desired properties, we exploit two simultaneous translations. This reduces the total amount of effort and complexity involved by tackling the problem in smaller parts.

The approach entails the construction of a semantic yardstick to which the results of the original translation are compared to. For this a deep embedding to the same host logic is used. To ensure the correctness of the original translator we take a case by case approach where we check the semantic equivalence of the deep and shallow representations of a specification. This equivalence proof is systematic and can be derived from the specification by a dedicated tool.

In effect, we have exchanged a lot of the formal reasoning necessary in translator verification into a software engineering problem. This can be especially beneficial with experimental languages and prototype tools.

A novel benefit stemming from the use of two embeddings is the manner in which the deep embedding can be simplified. The type system need not be represented in the deep embedding as long as it can be borrowed from the host logic and thus handled automatically in the shallow embedding.

References

[1] A. Azurat and I.S.W.B. Prasetya. A preliminary report on xmech. *UU-CS (Ext. rep. 2002-008). Utrecht, The Netherlands: Utrecht University: Information and Computing Sciences*, 2002.

[2] A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. *UU-CS (Ext. rep. 2002-007). Utrecht, The Netherlands: Utrecht University: Information and Computing Sciences*, 2002.

[3] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[4] Amy P. Felty, Douglas J. Howe, and Abhik Roychoudhury. Formal metatheory using implicit syntax, and an application to data abstraction for asynchronous systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16-99)*, volume 1632 of *LNAI*, pages 237–251, Berlin, July 7–10 1999. Springer.

[5] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In Graham M. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[6] Joni Helin and Pertti Kellomäki. Concern-based specification of distributed systems using behaviourally complete views. In Hong Mei Minhuan Huang and Juanjun Zhao, editors, *Proceedings of the International Workshop on Aspect-Oriented Software Development (WAOSD 2004), in conjunction with 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM2004)*, pages 74–78, September 2004.

[7] Leslie Lamport. TLA – temporal logic of actions. At URL `http://www.research.digital.com/SRC/tla/` on the World Wide Web.

[8] César Muñoz and John Rushby. Structural embeddings: Mechanization with method. In Jeannette Wing and Jim Woodcock, editors, *FM99: The World Congress in Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 452–471, Toulouse, France, September 1999. Springer-Verlag.

[9] The PVS home page at `http://pvs.csl.sri.com`, 2005.

[10] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at `http://www.csl.sri.com/shankar/PVSeval.ps.gz`.

[11] L. Zuck, A. Pnueli, Y.Fang, and B. Goldberg. Voc: A translation validator for optimizing compilers. In *Proceedings of the Internation workshop on Compiler Optimization meets Compiler Verification (COCV)*, 2002.

Incremental Confined Types Analysis

Michael Eichberg[1] Sebastian Kanthak[2] Sven Kloppenburg[3]
Mira Mezini[4] Tobias Schuh[5]

Department of Computer Science
University of Technology
Darmstadt, Germany

Abstract

Research related to alias protection and related concepts, such as, confined types and ownership types has a long tradition and is a promising concept for the design and implementation of more reliable and secure software. Unfortunately, the use of these concepts is not widespread as most implementations are proofs of concept and fall short with respect to the integration with standard software development tools and processes.

In this paper, we discuss an implementation of confined types based on Java 5 annotations. The contribution of this paper is twofold: First, we discuss the incrementalization of the confined types analysis and second, we present the integration of the analysis into Eclipse using the static analysis platform Magellan.

1 Introduction

Unintended aliasing is causing many kinds of problems. For example aliasing makes modular reasoning more difficult, as it is hard to reason about the effect of updating an object `o` when it is unknown which other objects also keep a reference to `o`.

Besides being a source of programming errors that can be detected when testing an application, unintended aliasing can also lead to security errors, which are hard to detect using standard development techniques. For example, when a reference to an object is passed to another object and, hence, an alias is created for the first object, then the alias can later on be used to update the first object in an unanticipated manner. In [19] a security breach caused

[1] Email: `eichberg@informatik.tu-darmstadt.de`
[2] Email: `kanthak@st.informatik.tu-darmstadt.de`
[3] Email: `kloppenburg@informatik.tu-darmstadt.de`
[4] Email: `mezini@informatik.tu-darmstadt.de`
[5] Email: `schuh@informatik.tu-darmstadt.de`

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

by a reference leaking bug in the JDK 1.1 is discussed (shown in Listing 1). In the JDK's implementation, each instance of a Java `Class` object holds an

```
1 public class Class {
2   private Identity[] signers;
3   public Identity[] getSigners() {
4     return signers;
5 } }
```

Listing 1: Class.getSigners() without Confined Types

array of signers (Line 2) that represents the principals under which the class acts. The problem is that the `getSigners` method returns a reference to the original `signers` array (Line 4). Hence, the attackers can then freely update the signatures based on their needs.

To solve the problems related to the creation of unintended aliases, we need means to enforce that important data structures can not escape the scope of a well defined protection domain. For example, to assure that the reference to the original signers array does not escape the declaring class. In [19], Vitek et Bokowski propose the concept of Confined Types to solve issues related to object aliasing.

In this paper, we present an incremental analysis for the confined types concept proposed in [19] and integrate this analysis into the incremental build process of the Eclipse IDE [7]. One goal of our work was compatibility with the Java language specification and existing tools. This was a major reason why we have chosen Confined Types [19]; using Java annotations we were able to simulate the necessary language extensions proposed by Vitek et Bokowski.

The main contribution of our work is to provide an implementation of the confined type checking that is tightly integrated with a standard software development environment and where the analysis exhibits a behavior that is indistinguishable from other (standard) compile time analyses. This fits well in the development philosophy supported by modern IDEs such as Eclipse, where the developer expects to see e.g., typing problems as soon as they emerge as the project evolves.

In general, we argue that — whenever possible — checking various program properties should be done by IDEs. This avoids bloated compilers and ensures that application-specific checkers can be introduced when needed. However, (re)checking the entire project after a change is prohibitively expensive w. r. t. the time required for the analysis. Hence, violations of the typing rules for confined types should be checked for incrementally.

In vein of these considerations, we have implemented the confinement rules defined in [19] using the open, extensible static analysis platform Magellan [9], which is tightly integrated into the Eclipse IDE [7]. By choosing Magellan and Eclipse as the underlying frameworks many issues related to tool adoption [1,10] are already solved. By building on top of Magellan, our analysis is automatically integrated with the incremental build process. Hence, the user will — after activation — perceive no difference between the checks carried out by the standard Java compiler and our analysis. This flattens the learning curve, as it is not necessary to learn how to use the tool, provided the developer is already familiar with Eclipse. Additionally, since we (re)use the standard Eclipse views to visualize errors no user interface related issues arise.

This paper is structured as follows: In the following, we first give an overview of confined types. After that, we introduce the static analysis platform Magellan on top of which we have build our analysis. We continue with a presentation of the implementation of the confined types analysis, and in particular, the issues related to the incrementalization of the analysis. After that, we evaluate our approach by using confined types in a large project. We conclude with a discussion of related work and a summary.

2 Confined Types

Confined Types were proposed by Vitek and Bokowski [19] as a machine checkable programming discipline that prevents leaks of sensitive object references. A motivation for their work was the security breach mentioned in the introduction (shown in Listing 1).

A possible solution to avoid the breach is a programming style that encourages the developers of classes with sensitive information to return a reference to a copy of the sensitive data, in our case a copy of the signers array. While programming styles cannot be enforced, using confined types ensures that none of the key data structures used in code signing escape the scope of their defining package.

For this purpose, types whose instances should not leave their defining package are marked as *confined*. *Confinement* ensures that objects of a confined type can only be accessed within a certain protection domain. A type is confined to this domain if all references to objects of that type originate from within the domain. Code outside the protection domain is never allowed to manipulate confined objects directly. In contrast to existing access control mechanisms in Java (such as the Java `private` keyword), confinement constrains access to object references rather than classes. It prevents class-based restrictions from being circumvented by casting the protected object to one of its unrestricted super-types.

In this paper, we describe an incremental analysis for the proposal in [19], integrated into the incremental build process of the Eclipse IDE. As proposed in [19], we also use Java packages as protection domains. Instead of the new modifiers, `confined` and `anon`, introduced in [19], we use the metadata facility (*annotations*) introduced in Java 5.0 and define two annotation types: `@confined` and `@anon`.

Listing 2 shows, how the code from Listing 1 can be rewritten using confined types. The annotation `@confined` is used with a class, whose objects should be confined to the containing package. In Listing 2, annotating `SecureIdentity` as `@confined` (Line 3) enforces references to `SecureIdentity` objects to be confined to the package `java.security`. Thus, code outside this package can never access instances of type `SecureIdentity`. Renaming the old `Identity` class to `SecureIdentity` and introducing a new `Identity` class (Line 4 – 8) preserves the functionality of the original interface.

```
 1 package java.security;
 2   abstract class AbstractIdentity { @anon equals(){...};  }
 3   @confined  class SecureIdentity extends AbstractIdentity { ... }
 4   public class Identity {
 5     SecureIdentity  target;
 6     Identity(SecureIdentity  t)  {  target  = t; }
 7     ...  // public  operations  on  identities ;
 8   }
 9   public class Class {
10     private SecureIdentity[]  signers;
11     public Identity[]  getSigners( ) {
12       Identity []  pub = new Identity[signers.length];
13       for  (int  i  = 0; i  < signers.length;  i++)
14         pub[i]  = new Identity(signers[i ]);
15       return pub;
16     }
17   }
```

Listing 2: Class.getSigners() using Confined Types

The `@anon` annotation enables confined types to safely use methods from unconfined types. Methods that do not reveal the current object's identity are marked as *anonymous* by annotating them with `@anon` to show this intention and to make this property checkable[6]. In Listing 2, the method `equals` in line 2 is marked with `@anon` to show that it never reveals the current instance's identity (`this`-reference). Therefore, `SecureIdentity` can safely extend `AbstractIdentity` and call `equals` on `this`, because no method marked `@anon` will breach the confinement.

The constraints in Table 1 and 2 are defined in [19] and define the semantics of `confined` and `anon`. Constraints in Table 1 restrict class and interface declarations (*C1*, *C2*), prevent widening (*C3*), hidden widening (*C4*, *C5*), and transfers from inside (*C6*) and outside (*C7*, *C8*) the protection domain. The rules defined in Table 2 constrain the usage of the self-reference `this` in method implementations, so that `this` is not revealed to code outside the method.

[6] Another possibility would be to infer the `@anon` property. But having it explicit as an annotation in the code serves as a documented design decision.

C1	A confined class or interface must not be declared public and must not belong to the unnamed global package.
C2	Subtypes of a confined type must be confined as well.
C3	Widening of references from a confined type to an unconfined type is forbidden in assignments, method call arguments, return statements, and explicit casts.
C4	Methods invoked on a confined object must either be non-native methods defined in a confined class or be anonymous methods.
C5	Constructors called from the constructor of a confined class must either be defined by a confined class or be anonymous constructors.
C6	Subtypes of `java.lang.Throwable` and `java.lang.Thread` may not be confined.
C7	The declared type of public and protected fields in unconfined types may not be confined.
C8	The return type of public and protected methods in unconfined types may not be confined.

Table 1
Constraints for confined types

A1	The reference `this` can only be used for accessing fields and calling anonymous methods of the current instance or for object reference comparisons.
A2	Anonymity of methods and constructors must be preserved in subtypes.
A3	Constructors called from an anonymous constructor must be anonymous.
A4	Native methods may not be declared anonymous.

Table 2
Constraints for anonymous methods

Using confined types as an extension to the Java type system, the programming style of returning only copies of sensitive data can be supported in such a way that once a type is marked as `@confined`, the safety of the program with respect to avoiding unintended reference leaking can be guaranteed.

3 Magellan

In this section, we discuss the static analysis platform Magellan. Magellan is a generic, extensible platform for static analyses, which is tightly integrated into the Eclipse IDE. The part of the architecture of Magellan relevant for this paper is depicted in Fig. 1. The types in the highlighted area (`Checker`, `ProblemsViewRE` and `Report`) are extended or used by classes of our confined types analysis. In the following, we briefly discuss the functionality of the central classes and interfaces and the interaction between them.

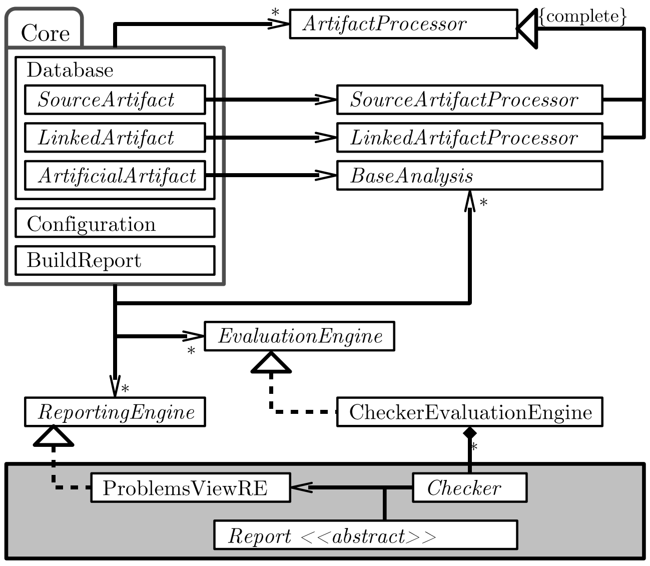


Fig. 1. Diagram of the main classes and interfaces

3.1 Source Artifact Processors

Source artifact processors create representations of program elements defined in the files of a project. The representations, which are appropriate for static analysis are called *source artifacts* and are stored in the database that is part of Magellan's core module; the database is basically a map that associates an Eclipse resource with artifacts generated by the processors.

The source artifact processor relevant for the confinement analysis uses the Java Bytecode Analysis Toolkit BAT [8] to create a quadruples[7] based representation of Java class files. The representation is similar to the *Jimple* representation of the Soot framework [18] and is essentially a register based representation of Java bytecode. A quadruples based representation facilitates many analyses, in particular data-flow analyses [17,18], when compared with a direct representation of Java bytecode as generated by other bytecode toolkits, such as e. g., BCEL [2].

3.2 Linked Artifact Processors

A linked artifact processor creates representations of resources that are not directly defined as part of the project, but which are relevant for the analysis of the project. E. g., classes in the Java runtime library are not defined as part of the project, but information about them is required by many analyses.

[7] Please note, the term *quadruple* and *3-address instruction* are used interchangeable [17, p. 479].

Representations generated by linked artifact processors are called *linked artifacts* and are also stored in the database. The linked artifact processor used by the confined types analysis processes the source artifacts generated from Java class files. The algorithm that this processor uses to determine the set of linked artifacts to add to the database is described next. The representations of all classes defined in libraries that are directly used in the implementation of the classes of the project are added. Next, the same process is recursively applied to each linked artifact added previously until every class used by any other class is added.

In particular, this algorithm ensures, that representations of the super types of every used type are made available. For illustration, assume that the only class in our project is the following:

```
class A { java.lang.Iterable l; }
```

In the first step, the algorithm adds a representation of the interface `Iterable` to the database — the type of the declared field. Further, the class `java.lang.Object` is added, since every class inherits from it. In the second step, the representations of `Object` and `Iterable` are analyzed. Since `Object` is the top-most type and `Iterable` does not extend any interfaces, no further classes need to be added. To reduce the size of the database, private methods and fields, as well as the methods' implementations are omitted.

3.3 Base Analyses

The program model generated by the source and linked artifact processors is enriched and rendered more precise by applying base analyses that exploit general-purpose program analysis techniques, e.g., class hierarchy, control-flow or data-flow analysis. Our confinement analysis uses the following two base analyses provided by Magellan: (a) the hierarchy analysis to make information about the super-/subtypes of a class directly available, and (b) an analysis to bring the quadruples representation in SSA form [6]. When the representation is in SSA form, the local variables' definition-use and use-definition information is directly available. This enables a straightforward implementation of the check that the `this` reference of a confined type is not passed to another object. For each value passed to another object or returned by the method, we have to check if the `this` reference is potentially assigned to it. To do so, we analyze the explicitly available use-definition information.

3.4 Evaluation Engines

Conceptually, an evaluation engine is a mediator between the Magellan core and a set of so-called checker modules. A checker module analyzes the models about the program, generated by the processors and base analyses, to derive higher-level information about the "correctness" of the program. The result of a checker is directly presented to the end user of a Magellan enabled IDE. For example, a result could be that a confinement rule [19] of a class is violated. The evaluation engine we are using for the confinement analysis enables to write checkers that directly work on the quadruples based representation. This evaluation engine provides a lightweight plug-in interface: Each checker must implement a small `Checker` interface to enable the evaluation engine (a) to determine the analyses and processors required by the checker and (b) to start the evaluation process.

3.5 Reporting Engines

A reporting engine displays the results of an analysis. During the evaluation of the database, reports that describe findings of the checkers are generated and passed to the reporting engines. Each reporting engine supports a specific reporting format. In our case, we use the simplest form of a report: a short descriptive text such as "*this* `must not be passed to another class`" associated with a particular artifact element. The reporting engine for this simple format uses the `Problems View` of Eclipse to display the generated reports. These reports consist of a short message, a severity level, a reference to the underlying resource and the specification of a source range to which the message refers.

3.6 The Magellan Core

The Magellan core is responsible for controlling the analysis process. The analysis process is triggered by an incremental or a full build. We will first describe the incremental build process. A high-level overview of the analysis process triggered by an incremental build is depicted in Fig. 2.

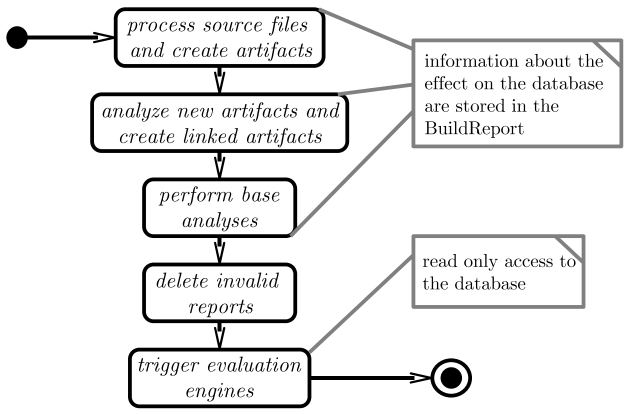

process source files and create artifacts
analyze new artifacts and create linked artifacts
perform base analyses
delete invalid reports
trigger evaluation engines
information about the effect on the database are stored in the BuildReport
read only access to the database

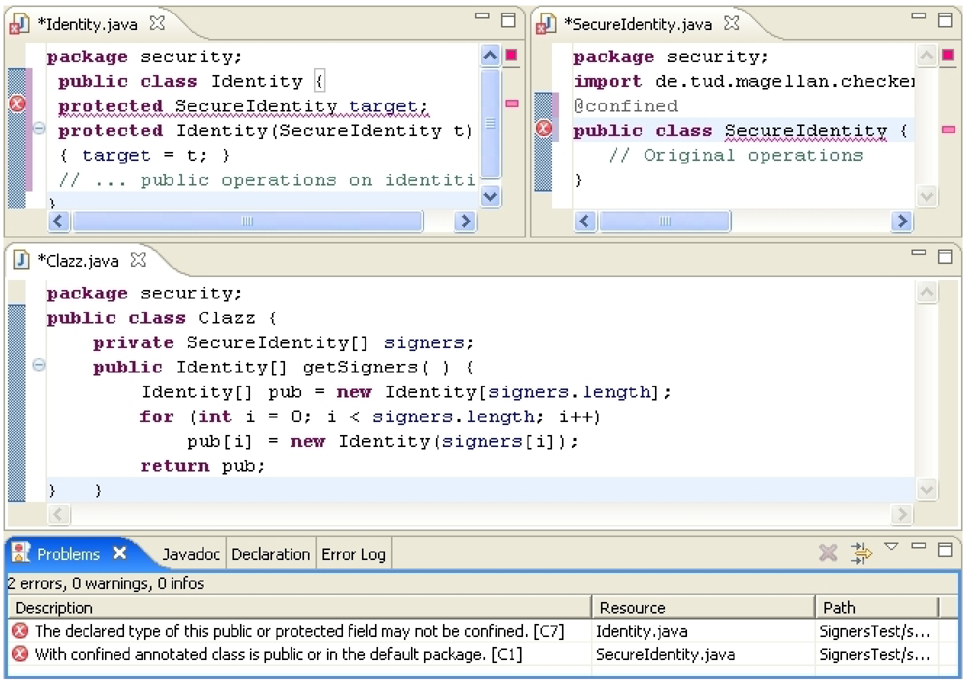
Fig. 2. Activity diagram of the build process

First, a `BuildReport` is created and used to record all changes to the database. The core uses the information passed to it (by Eclipse) to remove all artifacts from the database whose underlying resource has changed or was removed. The removed artifacts are added to the build report and are available until the end of the analysis process. The core passes each resource that has changed or which was added to all processors to obtain respective artifacts. When an artifact is returned, the core adds it to the database and to the build report. Second, the core passes the build report to the first linked artifact processor. The processor uses the information stored in the database and in the build report to determine the set of resources for which it needs to create linked artifacts. Third, the base analyses are executed. After performing all analyses, the build report also records the information about the artifacts where the analysis information has changed. Finally, the core calls the evaluation engines which then execute the registered checkers.

In case of a full build, the process is basically the same as described above, except that first the entire database is cleared and the source artifact processors are called for all files of the project. After that, the same steps are executed as in case of an incremental build.

4 Incremental Analysis

As stated in [19], checking the confinement rules is modular in the sense that each class can be analyzed separately. However, in addition to modularity and dynamic loading [19], our aim is to also support (a) continuous checking of confinement constraints during a programming task, and (b) IDE-Integration of the checking process with an integrated error reporting and source code navigation, as illustrated by the screenshot in Fig. 3.



```
package security;
 public class Identity {
 protected SecureIdentity target;
 protected Identity(SecureIdentity t)
 { target = t; }
 // ... public operations on identiti
}
```

```
package security;
import de.tud.magellan.checke
@confined
public class SecureIdentity {
   // Original operations
}
```

```
package security;
public class Clazz {
    private SecureIdentity[] signers;
    public Identity[] getSigners( ) {
        Identity[] pub = new Identity[signers.length];
        for (int i = 0; i < signers.length; i++)
            pub[i] = new Identity(signers[i]);
        return pub;
}    }
```

2 errors, 0 warnings, 0 infos

Description	Resource	Path
The declared type of this public or protected field may not be confined. [C7]	Identity.java	SignersTest/s...
With confined annotated class is public or in the default package. [C1]	SecureIdentity.java	SignersTest/s...

Fig. 3. Screenshot of Eclipse when using Confined Types

In such a setting, checking all constraints on all classes after every change is obviously prohibitive in terms of incremental build performance. However, determining which classes have to be reanalyzed after a set of arbitrary changes to the project's source code is non-trivial. For an example of how a small change can impact the confinement rules at a seemingly unrelated location, consider Listing 3.

```
 1 package x;
 2   public class X1 {
 3     @anon public void m() { /* ... */ }
 4   }
 5   public class X2 {
 6     public void m() { /* ... */ }
 7   }
 8
 9 package y;
10   public class Y extends X1 { } /* change: ... extends X2 */
11
12 package z;
13   @confined class Z extends Y { /* ... */ }
14   class W {
15     public void foo() {
16       Z z = new Z();
17       z.m(); /* will violate C4 after change */
18   } }
```

Listing 3: Indirect violation of confinement constraints

The example consists of Java classes in three different packages. Class `W` calls a method `m` on a confined class `Z`. $C4$ is satisfied because `Z` inherits `m` from class `X1` where it is declared anonymous. Now, let us assume that `Y` is changed to inherit from `X2` instead of `X1`. Since `X2` does not declare `m` as anonymous, the method call in Line 17 now violates constraint $C4$. Hence, a change in package `y` (which does not contain any confined or anonymous declarations) yields a confinement error in a class in package `z` that is neither a subtype nor a supertype of the changed class `Y`.

The example shows that when a class changes, it is not sufficient to only check classes in the same package / protection domain or all super-types and subtypes of the changed class. We therefore employ a more systematic approach to develop an incremental algorithm for checking the confinement rules.

Our checking algorithm is designed in two steps. First, given a list of classes that have been changed a set of classes is identified that must be reanalyzed to discover any new constraint violation and to remove any error message for constraints that are no longer violated. Next, the constraint rules are checked for all classes returned by the first step. Whenever a check fails an error report for the Eclipse problems view is created and presented to the user (see Fig. 3). Hence, after editing a source file the developer is immediately informed about constraint violations.

We regard all the constraints from Table 1 and Table 2 as predicates over classes, respectively over methods. For any class c, $C_i(c)$ is true, if and only if c satisfies C_i for any method m, $A_i(m)$ is true only if m satisfies the constraint A_i. Each predicate can be evaluated on its own, since the definitions of the constraints do not depend on each other. For example, for a class c to satisfy constraint $C4$ it suffices that methods called on confined types within c are declared as anonymous. Whether these methods, in turn, satisfy the constraints for anonymous methods is irrelevant for $C4$, though. The reason is that error messages are directly related to predicates that are not fulfilled. Violations of the constraints for anonymous methods will be displayed as separate errors when analyzing the respective methods.

Now we can state our problem as follows: Given a program, the predicate values for all its classes and methods, and a set of classes changed in the process of an incremental build, update the predicate values so that they reflect the program changes. This update process should be *correct* in the sense that it produces the same results as a whole-program analysis.

Since a constraint must only be reevaluated if some information it depends on has been invalidated by a program change we determine for each constraint the set of information it depends on.

Before doing so, we slightly modify the constraints $C2$ to $C2'$: "If a direct super-type of a type t is confined, t must be confined as well.", and $A2$ to $A2'$: "If a method m directly overrides an anonymous method, m must be anonymous as well." These modifications, while reducing the information on which the values of $C2$ and $A2$ predicates depend on, do not affect the semantics of the confined types: A program satisfies all the constraints from Table 1 and Table 2 if and only if it satisfies them with $C2$ and $A2$ replaced by $C2'$ and $A2'$.

We start our analysis by investigating the rules for anonymous methods, as defined in Table 2.

- $A1(m)$ depends on the anonymous attribute of all methods called on `this` inside m. These methods have been declared either in m's class or in a super-type of the latter. Hence, for any changed class c, $A1(m)$ must be reevaluated for any m in c or any of its subtypes.
- $A2'(m)$ depends on the anonymous attribute of the method overridden by m. Since such a method must be declared in a super-type of m's class, the same invalidation strategy as for $A1$ applies.
- Since calls to constructors from within a constructor can be seen as a special kind of method calls on `this`, we can treat $A3$ in the same way as $A1$.
- $A4$ does not depend on any non-local information. Thus, it suffices to reevaluate $A4$ on all methods of a changed class.

This leads to the following incremental algorithm for checking the constraints from Table 2. Whenever a type t changes, we have to reevaluate constraints $A1$–$A3$ on all subtypes of t (including t itself). Constraint $A4$ only has to be reevaluated for types that have been changed.

Next, the constraints in Table 1 are analyzed in the same way.

- $C1(c)$ only depends on information from the class c. Thus, for every c, which has changed, $C1(c)$ must be reevaluated.
- $C2'(c)$ depends on the confined attribute of all direct super-types of c. Thus, we have to reevalute $C2'(c)$ for any c that is a direct subtype of a changed class c'.
- $C3(c)$ depends on the confined attribute of the types used in widenings inside one of c's methods. The value of $C3(c)$ can change only if either c is changed (so that the list of widenings performed inside c has changed) or if the confined attribute of a type t that is used in a widening changes. For each such t, the following holds: t has been confined at some point (i. e., before or after the change), hence, t is defined within the same package as c. Therefore, for each class c whose confined attribute has changed $C3$ needs to be reevaluated for any class in the same package as a class c.
- $C4(c)$ depends on method calls in c where the static type of the receiver is confined. More specifically, it depends on the confined attribute of the method's declaring type and the method's anonymous attribute.

  Since the static receiver type is confined, it must be in the same package as the class that contains the method call. Thus, whenever the confined attribute of a type t changes, $C4(c)$ must be reevaluated for any class c in the same package as t to recheck all relevant method calls on t.

  Additionally, we have to reevaluate $C4$ when the anonymous attribute of the called method changes. This can happen indirectly as we have seen in the example from Listing 3. Thus, whenever a type t is changed we have to determine all classes that call a method on a confined subtype t' of t. Since a confined type can only be package visible, such a class must be in the same package as t'. For every confined subclass t' of t we check $C4(c)$ for all classes c in t''s package.
- The constraint $C5(c)$ considers constructor calls in constructors of confined classes. Since constructors are not inherited in Java, they have to be in the same class or in the direct superclass (can be called via `super(...)`). This implies that $C5$ depends only on the class itself and its superclass. When a class c is changed, we reevaluate $C5$ for c and all direct subtypes.
- $C6(c)$ depends on all super-classes of c. Thus, it suffices to reevaluate $C6$ for all subclasses of c whenever c is changed. As an optimization, we can ignore changes to c that do not change c's super-types.
- $C7(c)$ can change whenever the confined attribute of a type used in a public or protected field declaration of c changes. Since such a field type either was confined before the change or has become confined after the change, it has to be in the same package as c. Thus, whenever a type t changes $C7$

needs to be reevaluated for all classes in the same package as t.

- The constraint $C8(c)$ checks return types of methods that are declared as public or protected. The strategy for evaluating $C8$ is the same as for $C7$.

Given a set of files that have been changed, we process every constraint separately. For every changed class we compute the set of classes that have to be reanalyzed and then reevaluate the constraint against all classes in this set[8]. This process is correct even if multiple changes have been performed, because it analyzes the same classes that would have been analyzed if an incremental analysis had been performed after every change.

By definition, the rules for computing the set of classes to be checked after a change guarantee that a constraint is reevaluated if any information it depends on has been invalidated. Hence, the value of all predicates is the same as if they had been evaluated by performing a whole-program analysis. Thus, our incremental algorithm is correct. Regarding its efficiency, with the current rules we often have to reevaluate a constraint for all subtypes of some type. Obviously, this may be a very big set. Suppose, for example, that the class `Object` is changed somehow. Now, constraints $A1$–$A3$ for example have to be reevaluated for all subtypes of `Object` which essentially is every type.

A possible optimization is to use a call-graph analysis to reduce the reevaluations of constraints $A1$ and $C4$. This is because, we could determine all method call statements that are affected by a given change. For the change from Listing 3, for example, the call-graph analysis would tell us that the method called in Line 17 has changed and we could reevaluate $C4$ for this location. This avoids having to check constraints $A1$ and $C4$ for all classes in a package. The challenge, of course, is to make call-graph analysis incremental as the cost would be prohibitive otherwise and to make it fast enough to pay off compared to our current algorithm.

5 Performance Evaluation

The runtime complexity of static analyses is an important obstacle for their widespread adoption; performance is especially crucial for an integration into the build process. To assess our analysis in this respect, we measured its runtime while refactoring the Java runtime library to implement the suggestions made in [12]. The experiment was conducted on a dual Xeon 3.0Ghz workstation with 2GB RAM running Windows XP and the Sun Java 5 JDK.

We edited the "`public`" part of the Java 5 runtime library (rt.jar) delivered with the Sun JDK, which consists of 4992 classes in `java.*`, `javax.*` and `org.*`. Furthermore, 441 classes were added to the database by the linked artifact processor for classes in `sun*` and `com.sun*`; these classes are used in the implementation of the public classes.

[8] For simplicity, we just compute the union of all these sets and check all constraints against every class in this set.

To keep the artifacts, the hierarchy information and the results of the confinement analysis in memory ≈ 85MB are required. The overall time for the first analysis process (full build, without confinement annotations) of the project is 46.5 seconds; the supporting analyses require 45.7 seconds and the analysis of the confined types (`Confinement Analysis`) 0.7 seconds.

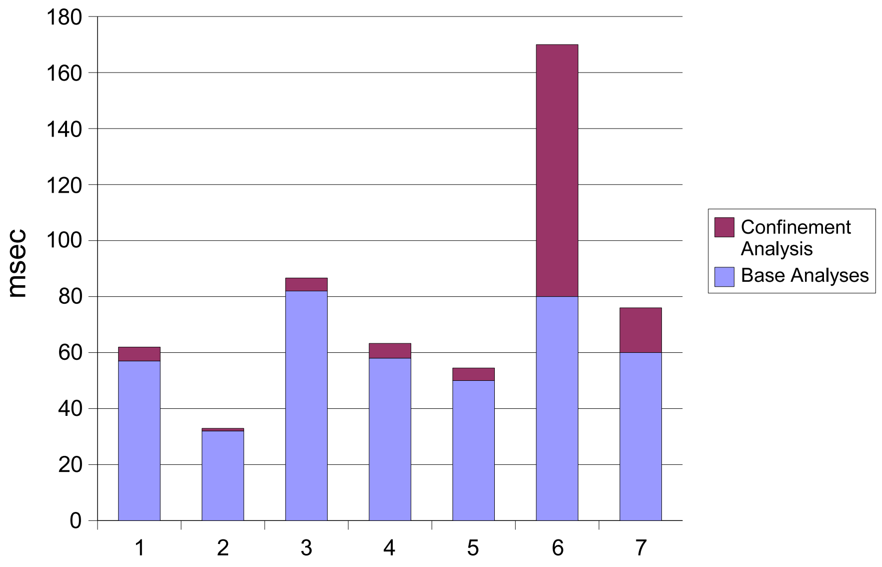


Fig. 4. Incremental build times (msecs)

The time required to perform the analysis during incremental builds is shown on the y-axis in Fig. 4. The numbers on the x-axis are identifiers for different refactorings that we performed:

(i) A complex class (1578 LOC) is changed; the change does not affect confined types.

(ii) The implementation of an anonymous method is changed, whose declaring class does not have confined subclasses.

(iii) A method in a confined class is changed; the change does not violate any confinement constraints.

(iv) A method is annotated as anonymous which is inherited and used by a confined subclass.

(v) The annotation added in the previous case is removed to force the creation of an error message.

(vi) A class is annotated as confined in a package that previously did not define any confined types.

(vii) The confined annotation of the most recently annotated class in a package is removed.

The results show that in case of an incremental build the time required to perform the necessary analyses is in general less than 200 milliseconds[9]. Further, the additional amount of memory required is at most 85MB. These results indicate that it is feasible to run the confinement analysis along with the incremental build process.

6 Related Work

When dealing with aliasing, four categories of work are considered [14]: detection, prevention, control and advertisement of aliasing. The works we are interested in, mostly fall under the category of prevention and control.

The notion of alias protection for object-oriented languages was introduced by Hogg [13] in order to enable modular reasoning for groups of classes. These groups are called *islands* and ensure the restriction of aliasing to classes on the island. Hogg differentiates between static and dynamic aliases. Static aliases are aliases via instance variables and dynamic aliases are those via parameters or local variables. Static aliasing can lead to undesired side effects in later invocations of the aliased object. Dynamic aliases were seen as unproblematic, because they disappear at the end of the execution of the method in which they are defined. Means to control static aliasing were introduced with islands. Islands are the transitive closure of a set of objects accessible from a *bridge* object. A bridge object is the sole access point to a set of instances that make up an island.

To ensure that no static aliases are created from outside the island to objects on the island, the methods of the bridge object are restricted. Only methods with parameters and return values that either do not modify the state of the system, or have only parameters and return values that have at most one static alias are allowed. This avoids the creation of unwanted aliases. For example, a return value of a method can be tagged with *unique* to state that exactly one reference to its value exists. The value can only be assigned to other variables, if the original reference is released.

The full encapsulation of aliases of this approach is too restrictive for many common design idioms used in OO programming. E. g., no object could be a member of two collections simultaneously if either collection was fully protected against aliases. In this case, one collection would be an island, prohibiting that references to its members show up outside the island.

In [15], Noble et al. present a more flexible approach to control aliasing when compared with *islands*. The approach taken by Noble et al. is to enable aliasing by introducing explicit aliasing modes. The authors differentiate between the *representation* of an object, which corresponds to its fields, and *arguments*, which are parameters to methods of the object. The representation of objects should only be accessible via the object's interface, e. g., in Java

[9] Please note that the automatic parallelization of the artifact processors reduces the required time for processing the source files by $\approx 35\% - 40\%$ when compared to a single CPU configuration.

fields would have to be marked as `private` and aliases to them should not be returned via getter methods. The state of the object should only depend on arguments with an immutable state. If the state of the object was dependent on the mutable part of arguments to its methods, the state of the object could be changed by changing the state of the arguments long after the call, by-passing the objects interface. The approach uses tags to annotate types and enables the compiler to enforce the restrictions mentioned on the creation of aliases. A formalization of this model is discussed by Clarke et al. [5]. Even though both approaches enable flexible alias control, they are designed for a language without inheritance or subtyping.

A variant of ownership types is used by Boyapati et al. [3] to prevent data races and deadlocks by partitioning locks into a fixed number of equivalence classes and specifying a partial order among these equivalence classes. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in descending order. Ownership types are used to ensure that that the locks that protect an object also protect its encapsulated objects.

The approach of Clarke et al. [4] implements a confinement checker for Java to solve the domain specific problem of passing a `this` reference from one Enterprise Java Bean component to another component. In EJB access to the internal objects implementing each Bean must be prevented, and access to the Bean is permitted only through the container generated wrapper. While confined types are a generic solution to control aliasing, Clarke et al.'s approach solves an EJB specific problem.

The work of Fong [11] describes how to translate the notion of confinement, which is formulated for static analysis of Java source code, to dynamic analysis of Java Bytecode. The approach retains the confinement annotations made in the source code at bytecode level. This enables link time checks of confinement rules. It also describes a form of secure cooperation between mutually suspicious code units, where, for example, a resource object can be shared between two untrusting modules while ensuring its confinement to a given domain. The implementation extends the runtime of the Pluggable Verification Modules of the Aegis Research JVM. Our approach uses static analysis to ensure the confinement properties at compile time and to immediately inform the user of confinement violations.

In [20], the notion of confined types is formalized in the context of Featherweight Java (FJ). In FJ, confined types are extended to confined instantiations of generic classes.

Reverse engineering approaches to the detection of aliasing are described in [12,16]. Kacheck/J [12] is a tool to infer confinement in Java code and was used to test the thesis that all package-scoped classes in Java programs should be confined. About 25% of the classes of their benchmark suite were confined anyway and 45% could be refactored to be confined just by changing visibility modifiers. These numbers are supported by the findings of Potanin et al. [16]. They presented metrics of uniqueness, ownership and confinement by analysing snapshots of Java program's object graphs and found that a third of all objects were strongly confined.

7 Summary & Future Work

In future work, we will extend the analysis to implement confined types with support for generic data types. This would relax the restrictions now posed on the use of confined types as it enables putting confined types in containers, which are parameterized using the confined type. Further, we will add support for a more flexible definition of protection domains to broaden the range of use of the confinement analysis; e. g., to check Enterprise Java Beans for correctly confining `this` to the scope of the bean.

In this paper we have discussed an implementation of an incremental confinement analysis realized as an Eclipse plug-in that makes use of the static analysis platform Magellan. As the performance figures show, the overhead when always executing the analysis along with the incremental build process is low enough to be able to use confined types in day-to-day usage. Further, using Magellan we were able to overcome the identified tool adoption barriers while being able to focus on the implementation of the analysis.

The confined types analysis plug-in is freely available at:
`www.st.informatik.tu-darmstadt.de/Magellan`

References

[1] Balzer, R., J. Jahnke, M. Litoiu, H. A. Müller, D. B. Smith, M.-A. Storey, S. R. Tilley and K. Wong, *3rd international workshop on adoption-centric software engineering*, in: *Proceedings of ICSE 2003*.

[2] *Bcel* (2005).
URL `http://jakarta.apache.org/bcel/`

[3] Boyapati, C., R. Lee and M. Rinard, *Ownership types for safe programming: preventing data races and deadlocks*, in: *Proceedings of OOPSLA 2002*, 11.

[4] Clarke, D., M. Richmond and J. Noble, *Saving the world from bad beans: deployment-time confinement checking*, in: *Proceedings of OOPSLA 2003* (2003).

[5] Clarke, D. G., J. M. Potter and J. Noble, *Ownership types for flexible alias protection*, in: *Proceedings of OOPSLA '98* (1998).

[6] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems **13** (1991).

[7] *Eclipse 3.1* (2005).
URL `http://www.eclipse.org`

[8] Eichberg, M. and C. Bockisch, *Bat* (2005).
URL http://www.st.informatik.tu-darmstadt.de/BAT

[9] Eichberg, M. and C. Bockisch, *Magellan* (2005).
URL http://www.st.informatik.tu-darmstadt.de/Magellan

[10] Favre, J., J. Estublier and R. Sanlaville, *Tool adoption issues in a very large software company*, in: *Proceedings of ICSE 2003*, 2003.

[11] Fong, P. W. L., *Link-time enforcement of confined types for jvm bytecode*, in: *To appear in Proceedings of PST'05*, 2005.

[12] Grothoff, C., J. Palsberg and J. Vitek, *Encapsulating objects with confined types*, in: *Proceedings of OOPSLA 2001* (2001).

[13] Hogg, J., *Islands: aliasing protection in object-oriented languages*, in: *Proceedings of OOPSLA '91* (1991).

[14] Hogg, J., D. Lea, A. Wills, D. deChampeaux and R. Holt, *The geneva convention on the treatment of object aliasing*, SIGPLAN OOPS Mess. **3** (1992), pp. 11–16.

[15] Noble, J., J. Vitek and J. Potter, *Flexible alias protection*, in: *Proceedings of ECCOP '98* (1998).

[16] Potanin, A., J. Noble and R. Biddle, *Checking ownership and confinement*, Concurrency and Computation: Practice and Experience **16** (2004).

[17] Scott, M. L., "Programming Language Pragmatics," Morgan Kaufmann, 2000.

[18] Vallée-Rai, R., E. Gagnon, L. J. Hendren, P. Lam, P. Pominville and V. Sundaresan, *Optimizing java bytecode using the soot framework: Is it feasible?*, in: *Proceedings of CC 2000* (2000).

[19] Vitek, J. and B. Bokowski, *Confined types in java*, Software Practice and Experience **31** (2001).

[20] Zhao, T., J. Palsberg and J. Vitek, *Lightweight confinement for featherweight java*, SIGPLAN Not. **38** (2003).

LDTA 2006 Preliminary Version

Scannerless Boolean Parsing

Adam Megacz

Computer Science
UC Berkeley

Abstract

Scannerless generalized parsing techniques allow parsers to be derived directly from unified, declarative specifications. Unfortunately, in order to *uniquely* parse existing programming languages at the character level, disambiguation extensions beyond the usual context-free formalism are required.

This paper explains how scannerless parsers for *boolean grammars* (context-free grammars extended with intersection and negation) can specify such languages unambiguously, and can also describe other interesting constructs such as indentation-based block structure.

The `sbp` package implements this parsing technique and is publicly available as Java source code.

Key words: boolean grammar, scannerless, GLR

1 Introduction

Although scannerless parsing[2] was first introduced in [1], it was not practical for general use until combined with the Lang-Tomita Generalized LR parsing algorithm [2,3] by Visser [7]. Unfortunately, the context-free grammars for most programming languages are ambiguous at the character level, which motivated the introduction of six empirically-chosen disambiguation constructs: FOLLOW, REJECT, PREFER, AVOID, ASSOCIATIVITY, and PRECEDENCE.

2 Conjunctive and Boolean Grammars

Conjunctive grammars [8] augment the juxtaposition $(\cdot)$ and language-union $(\,|\,)$ operators of context-free grammars with an additional language-intersection (&) operator. Boolean grammars [10] further extend conjunctive grammars by

[1] Email: `megacz@cs.berkeley.edu`
[2] also called “lexerless” or “character-level”

permitting the language-complement operator (~) to be used, subject to some basic well-formedness constraints[3].

It should be noted that Visser arrives at a similar result from the opposite direction by using REJECT productions, which act as conjunction with a negation. The paper goes on to reconstruct simple negation as well as intersection in terms of this negated-conjunction primitive, even noting that "this feature can give rise to as yet unforeseen applications."[7]

3 Disambiguating with Boolean Constructs

We now examine each of the disambiguation constructs and explain how to recast it as a boolean expression.

The PREFER and AVOID attributes effectively make a context-sensitive choice between ambiguous parsings, thus turning an ambiguous context-free grammar into an *un*ambiguous context-*sensitive* grammar. We can replace PREFER and AVOID with an ordered-choice operator (`>`), which is a metagrammatical abbreviation for intersecting the lower-priority expression with the complement of the higher-priority expression [4]

The ASSOCIATIVITY and PRECEDENCE features cannot be expanded into simple context-free expressions when they span multiple nonterminals, as in example 4.5.11 of [5]. In this example, addition and multiplication expressions are defined for the real numbers (`R`) and natural numbers (`N`). A subsumption production (`N` $\rightarrow$ `R`) is included, but operations on the natural numbers assume higher priority than corresponding operations on the reals. This sort of rich priority specification can be expressed in a manner similar to the ordered-choice operator: expressions are intersected with the complement of all higher-priority expressions, even those which involve productions from multiple nonterminals.

Uses of the REJECT attribute can be trivially translated into intersection with the complement of the rejected expression.

A FOLLOW restriction can be written as a boolean expression if one considers *character boundaries* (pairs of adjacent characters) as input tokens. From this perspective, a FOLLOW restriction amounts to intersecting an expression with the set of all strings ending with a valid follow-boundary.

4 SBP: a Scannerless Boolean Parser

The `sbp` package is an implementation of the Lang-Tomita Generalized LR Parsing Algorithm [2,3], employing Johnstone & Scott's RNGLR algorithm [13] for handling ϵ-productions and circularities.

[3] for example, a nonterminal cannot be defined to produce exactly its own complement

[4] for example, `a > b` expands to `a | (b & ~a)`

The input alphabet for `sbp` is typically the set of individual Unicode characters, though any topological space[5] can be used. An interesting consequence is that `sbp` can parse sentences constructed from non-discrete alphabets.[6]

The parser's grammars are built programmatically and can be manipulated and through a simple API. A sample metagrammar is included; it supports alternation (`|`), intersection (`&`), complement (`~`), intersect-with-complement (`&~`), subexpressions (`()`), regular expressions (`*`, `+`, `?`), repetition with a separator (`*/`, `+/`), maximal character repetition (`++`, `**`), ordered choice (`>`), promotion operators (as in [12]), character ranges (`[a-z]`), and whitespace insertion (`/ws`).

5 Examples

5.1 Dangling Else

A classic example of grammatical ambiguity is the so-called "*dangling else*" construct [18]. Straightforward application of negation and intersection can exclude expressions with a trailing `else` clause from the `then`-branch of an `if` statement, leaving the intended interpretation as the only valid parse:

```
Expr   = "if" "(" Expr ")" IfBody
       | ...

IfBody = Expr              "else" Expr
       | Expr   &  ~([~]* "else" Expr)
```

The complement of the empty character class (`[~]`) is an idiom used to match any character.

5.2 Indentation Block Structure

Besides disambiguation, boolean grammatical constructs have an number of other applications. The following example parses a language with indentation-based block structure by imposing a well-formedness constraint on blocks. The technique employed was inspired by [11].

We begin with the grammar for a simple fragment of a C-like language. The grammar uses conjunction with a negated term to exclude identifiers whose names happen to be keywords, just as in [6].

```
Statement = Expr "()"              ident       = [a-z]++ & ~keywords
          | "while" Expr block     keywords    = "while" | "if"

Expr      = ident
          | [0-9]++
```

We can now use boolean language operations to impose additional struc-

[5] one for which the $\cup$, $\cap$, $\sim$ operators and the $\subseteq$ (or simply $=$) test are supplied

[6] although we have not yet found a practical use for this capability

ture. We will do this by defining a nonterminal for syntactic blocks, and intersecting it with another production which requires that *no line in a block can be indented less than the first line.* Lastly, we use the ordered choice operator to prefer "tall" (left-associative) `BlockBody` productions.

```
indent     = " "*
outdent    = " "  outdent " "
           | " "  [~]*    "\n"

block        = "\n" indent  BlockBody
            &~ "\n" outdent [~ ] [~]*

BlockBody    = Statement
             > Statement BlockBody
```

The `block` rule matches code blocks which start a new line. The rule requires a newline, followed by some number of spaces, followed by a `BlockBody`. This production is intersected with a well-formedness production: the newline must not be followed by an `outdent`.

Similar to the sort of rule used to match balanced parentheses, the `outdent` rule matches any text which begins with indentation and also contains some other (disjoint) instance of indentation which is *shorter* than the first instance. In the context of the `block` production, this would describe any block containing a line with indentation less than that of the first line in the block.

6 Related Work

The original scannerless generalized parser, `sglr`[5] was designed as an improved parser for the ASF+SDF[4] framework. Dparser [17] is an implementation of the GLR algorithm in ANSI C, with support for most of Visser's disambiguation rules.

Several GLR parsers are available which require a tokenizer.[7] These include Elkhound[14], and the GLR extensions to `bison`.

Parsing Expression Grammars (PEG)s[15] include a limited form of intersection and complement, and the corresponding algorithm [16] is effective at parsing character-level grammars. However, many interesting context-free grammars are not PEGs, and cannot be parsed this way.

7 Future Directions

The current implementation is written in Java. It generates parse tables (which can be saved and restored), but currently only provides support for

[7] some can be used as "character level" parsers, but lack the disambiguation capabilities necessary to parse most programming languages at this level

interpreting these tables. Emitting compilable source code equivalent to parsing from these tables will be an important step in improving the performance of `sbp`.

Like the `sglr` parser, `sbp` deliberately excludes support for semantic actions, preferring to keep grammar definitions implementation-language-neutral. One consequence is that parsing requires space which is linear in the input, since the entire parse tree (modulo portions removed using the drop operator) must be constructed before any part of it can be consumed. An important future direction is the possibility of constructing *lazy parse forests* which can be incrementally consumed and discarded by a process running concurrently with the parser.

8 Availability

The source code for `sbp` is available under the terms of the BSD license, at `http://research.cs.berkeley.edu/project/sbp/`.

References

[1] Daniel J. Salomon and Gordon V. Cormack. *Scannerless NSLR(1) parsing of programming languages.* SIGPLAN '89, pp 170-178. ACM Press, 1989.

[2] Lang, Bernard. *Deterministic Techniques for Efficient Non-deterministic Parsers.* Automata, Languages and Programming, Springer, 1974.

[3] Tomita, M. (1987). *An efficient augmented-context-free parsing algorithm.* Computational Linguistics, 13(1-2), 31–46.

[4] A. van Deursen, J. Heering and P. Klint (eds.), *Language Prototyping: An Algebraic Specification Approach*, AMAST Series in Computing, Volume 5, World Scientific, September 1996

[5] Visser, Eelco. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

[6] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. *Disambiguation filters for scannerless generalized LR parsers.* In *Compiler Construction (CC'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[7] Visser, E. (1997b). *Scannerless generalized-LR parsing.* Technical Report P9707, Programming Research Group, University of Amsterdam.

[8] Okhotin, Alexander. *Conjunctive Grammars.* Journal of Automata, Languages and Combinatorics 6(4): 519-535 (2001)

[9] Okhotin, Alexander. *LR Parsing for Boolean Grammars.* Developments in Language Theory 2005: 362-373.

[10] Okhotin, Alexander. *Boolean Grammars.* Developments in Language Theory 2003: 398-410.

[11] Okhotin, Alexander. *On the existence of a Boolean grammar for a simple procedural language.* Proceedings of AFL 2005.

[12] Johnstone, Adrian and Scott, Elizabeth. *Constructing reduced derivation trees.* University of London CSD-TR-97-27 (1997)

[13] Johnstone, Adrian and Scott, Elizabeth. *Generalised reduction modified LR parsing for domain specific language prototyping.* Proc. 35th Annual Hawaii International Conference On System Sciences (HICSS02), IEEE Computer Society, New Jersey, (January 2002).

[14] Scott McPeak and George C. Necula. *Elkhound: A Fast, Practical GLR Parser Generator.* Proceedings of Conference on Compiler Constructor (CC04), April 2004.

[15] Bryan Ford. *Packrat Parsing: Simple, Powerful, Lazy, Linear Time.* International Conference on Functional Programming, October 4-6, 2002, Pittsburgh

[16] Bryan Ford. *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.* Symposium on Principles of Programming Languages, January 14-16, 2004, Venice, Italy.

[17] http://dparser.sourceforge.net/

[18] Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language, Second Edition* Prentice Hall, Inc., 1988. ISBN 0-13-110362-8

A Domain-Specific Language for Generating Dataflow Analyzers

Jia Zeng [2]

Department of Computer Science
Columbia University, New York

Chuck Mitchell [3]

Microsoft Corporation
Redmond, Washington

Stephen A. Edwards [4]

Department of Computer Science
Columbia University, New York

Abstract

Dataflow analysis is a well-understood and very powerful technique for analyzing programs as part of the compilation process. Virtually all compilers use some sort of dataflow analysis as part of their optimization phase. However, despite being well-understood theoretically, such analyses are often difficult to code, making it difficult to quickly experiment with variants.

To address this, we developed a domain-specific language, Analyzer Generator (AG), that synthesizes dataflow analysis phases for Microsoft's Phoenix compiler framework. AG hides the fussy details needed to make analyses modular, yet generates code that is as efficient as the hand-coded equivalent. One key construct we introduce allows IR object classes to be extended without recompiling.

Experimental results on three analyses show that AG code can be one-tenth the size of the equivalent handwritten C++ code with no loss of performance. It is our hope that AG will make developing new dataflow analyses much easier.

Key words: Domain-specific language, Dataflow analysis, dynamic class extension, compiler, Phoenix compiler framework

[1] Edwards and his group are supported by an NSF CAREER award, an award from the SRC, and from New York State's NYSTAR program.
[2] Email: `jia@cs.columbia.edu`
[3] Email: `chuckm@microsoft.com`
[4] Email: `sedwards@cs.columbia.edu`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Modern optimizing compilers are sprawling beasts. GCC 4.0.2, for example, tips the scales at over a million lines of code. Much of its heft is due simply to its many features: complete support for a real-world language, a hundred or more optimization algorithms, and countless back-ends. But the intrinsic complexity of its internal structures' APIs and the verbosity of its implementation language are also significant contributors.

We address the latter problem by providing a domain-specific language, AG for "Analyzer Generator," for writing dataflow analysis phases in Microsoft's Phoenix compiler framework. Experimentally, we show functionally equivalent analyses coded in AG can be less than one-tenth the number of lines of their hand-coded C++ counterparts and have comparable performance.

Reducing the number of lines of code needed to describe a particular analysis can reduce both coding and debugging time. We expect our language will make it possible to quickly conduct experiments that compare the effectiveness of various analyses. Finally, by providing a concise language that to allows analyses to be coded in a pseudocode-like notation mimicking standard texts [1], compiler students will be able to more quickly code and experiment with such algorithms.

One contribution of our work is a mechanism for dynamically extending existing classes. In writing a dataflow analysis, it is typical to want to add new fields and methods to existing classes in the intermediate representation (IR) in the analysis. Such fields, however, are unneeded after the analysis is completed, so we would like to discard them. While inheritance makes it easy to create new classes, most object-oriented languages do not allow existing classes to be changed. The main difference is that we want existing code to generate objects from the new class, which it would not otherwise do.

The challenge of extending classes is an active area of research in the aspect-oriented programming community [7], but their solutions differ from ours. For example, the very successful AspectJ [6] language provides the intertype declarations that can add fields and methods to existing classes. Like ours, this technique allows new class fields and methods to be defined outside the main file for the class, it is a compile-time mechanism that actually changes the underlying class representation, requiring the original class and everything that depends on it to be recompiled. In AG, only the code that extends the class must be recompiled when new fields are added.

MultiJava [3] provides a mechanism that is able to extend existing classes without recompiling them, much like our own, but their mechanism only allows adding methods, not fields, to existing classes.

In AG, we provide a seamless mechanism for adding annotations to existing IR classes. In AG code, the user may access such added fields with the same simple syntax as for fields in the original class. Adding such fields does not require recompiling any code that uses the original classes.

We implemented our AG compiler on top of Microsoft's Phoenix, a framework for building compilers and tools for program analysis, optimization, and testing. Like the SUIF system [13], Phoenix was specifically designed to be extensible and provides the ability, for example, to attach new fields to core data types without having to recompile the core. Unfortunately, implementing such a facility in C++ (in which Phoenix is coded) has a cost both in the complexity of code that makes use of such a facility and in its execution speed. Experimentally, we find the execution speed penalty is less than factor of four and could be improved; unfortunately, the verbosity penalty of using such a facility in C++ appears to be about a factor of ten. Reducing this is one of the main advantages of AG.

2 Related Work

The theory of dataflow analysis is well-studied. Kildall [8] was one of the first to propose a unified lattice-based framework for global program analysis. Later, Kam and Ullman [5] addressed the iterative approach and made the theory more concrete.

Wilhelm [12] notes that there are many generic theories for dataflow analysis, but few tools are built on these theories and even fewer are widely accepted. One big reason is the lack of a standard mid-level program representation. We expect the Phoenix compiler framework to address this problem, at least for object-oriented imperative languages. Another reason for the lack of tools is their complexity. Thus the focus of our work is to provide a simple language and tool for writing dataflow analyses.

Tjiang's Sharlit [10] is a tool for building iterative dataflow analyzers and optimizers. It is built on the SUIF [13] generic compiler construction framework. However, Sharlit did not introduce a new language. It uses C++ and provides some APIs, much like the Phoenix environment, and its focus was mostly on its efficiency, not its simplicity. While it makes an implementation of an analysis much more modular, it remains difficult to use.

A few tools require an explicit definition of the lattice used in dataflow analysis. Examples include Alt and Martin's PAG [2], Venkatesh and Fischer's SPARE [11], and the flexible architecture presented by Dwyer and Clarke [4]. PAG is well-known and has been used in industry. There are many similarities between AG and PAG: both use basic blocks and unchanged-pre-condition checking to improve the speed of the generated analyzer. Both provide a "set" data type. Unlike AG, PAG requires the user to specify the lattice used during analysis, which provides more optimization choices, like widening and narrowing, and makes it easier to verify the algorithm's correctness, but this makes PAG descriptions larger and more complex.

Some tools specifically address interprocedural analysis, such as Yi and Harrison's auto-generation work [14]. We focus only on intraprocedural analysis, although many of our ideas should carry over to inter-procedural problems.

3 The Design of AG

AG is a high-level language that provides abstractions to describe iterative dataflow analyses. The AG compiler translates an AG program into C++ source and header files, which are then compiled to produce a Dynamically-Linked Library (DLL) file. (Figure 1) This DLL can then be plugged in to the Phoenix compiler and invoked just after a program is translated into Phoenix's Middle Intermediate Representation (MIR).

Our generated plug-in extends IR objects to collect information and invokes a traversal that is part of the Phoenix framework to perform iterative analysis. This traversal function invokes computations defined in the AG program.

We follow the classical dataflow analysis approach. An AG program implicitly traverses the control-flow graph of the program and considers a basic block at a time. Inside each block, the analysis manipulates its constituent instructions and operands. We thus chose to make blocks, instructions, and operands basic objects in AG. Phoenix, naturally, already has such data types, but AG makes them easier to uses since our language has a deeper understanding of them.

One of the main contributions of AG is the ability to add attributes and computations to these fundamental data types. This facility relies on mechanisms already built in to Phoenix, but because of the limitations of C++, making use of such mechanisms is awkward and tedious to code. AG makes it much easier.

To simplify the description of computation functions, we included new statements in AG such as *foreach* and data-flow equations like those found in any compiler text. We also introduced a *set* data type since data collected during dataflow analysis usually takes the form of sets.

AG relies on the Phoenix Traverser class. This is an iterative traverser that does not guarantee boundedness. See Nielson and Nielson [9] for a discussion of the issues in guaranteeing boundedness.

4 The AG Language

The AG language is designed for dataflow analysis. It provides abstractions for the common features of iterative intraprocedural analysis. For user convenience and adaptability, we chose a syntax similar to that of C++ and added a variety of new statements and constructs.

4.1 Program Structure

Figure 2 shows the structure of a typical AG program to describe an analyzer. It defines a new, named phase, extends a number of built-in Phoenix classes with new fields and methods to define what information to collect, and finally defines a transfer function for the dataflow analysis.

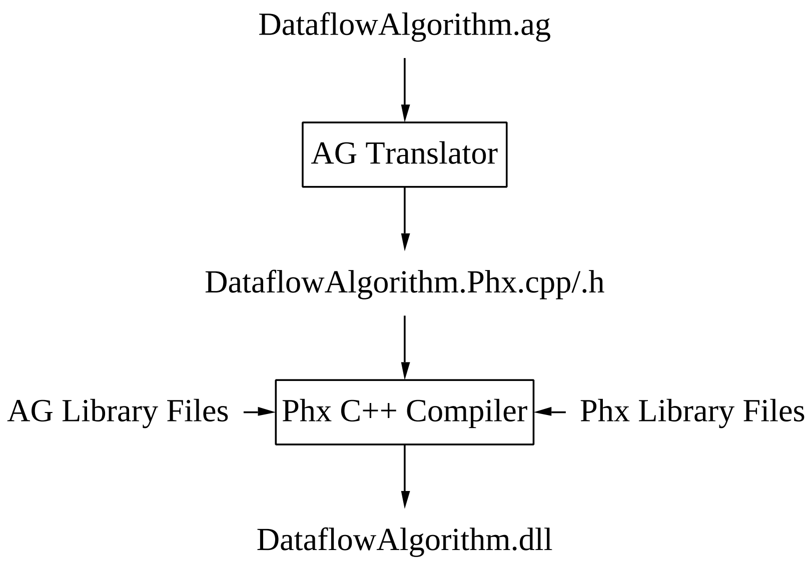


Fig. 1. The operation of the AG framework

```
Phase name {
    extend class name {
        field declarations...
        method declarations...
        void Init() { . . . }
    }
    ⋮
    type TransFunc(direction) {
        Compose(N) { . . . }
        Meet(P) { . . . }
        Result(N) { . . . }
    }
}
```

Fig. 2. The structure of an AG program

An *extend class* defines a new IR class that uses the Phoenix dynamically extensible IR class system. New fields and methods declared in an extend class are added as new class members. The user may directly refer to them as if they were members of the original class (our compiler identifies such fields and generates the appropriate Phoenix code to access and call members of such extended classes). Notice the methods declared in an extend class are "private," i.e. they can only be applied to the corresponding extend object, or in other methods declared under the same extend class. Currently, we only support extending Block, Instr, and Opnd classes.

In each extend class, the Init method behaves (and is executed as) an initializer just after the constructor for the extended class.

Each phase has a single TransFunc that defines the return type and iteration direction (backward or forward) of the analyzer and, more importantly, the equations applied during the analysis. The body of a TransFunc may define functions, especially three reserved functions: Compose, Meet, and Result. Compose and Meet functions are applied when the traverser iterates on every blocks. The Compose function defines the computation inside a block using global data. The Meet function defines the computation performed between blocks, i.e., to merge data from the exit of the predecessor to the entry of the successor. The Result function defines operations to be performed just after the iteration. It usually propagates information to the objects that make up the blocks, such as instructions. Other functions may be declared in the TransFunc; they can be called by the three reserved functions or each other.

The user may embed arbitrary C++ code in the body of these methods. Such code segments are transparent to AG compiler, which simply includes them verbatim in the generated code.

4.2 AG Syntax

We derived the syntax of AG from C++. We present its complete syntax in the appendix; Table 1 provides a summary. Below, we provide some details about its design.

Set is a data type similar to *set* in the C++ standard library. It can only apply to the reserved classes and actually refers to a set of IDs. For example, "Set<Instr>" will be translated into a bit-vector mapped on IDs of instructions in implementation. The *Map* type is similar.

During the analysis, the most relevant data are those with information for the entry and exit points of each block, so we introduced the *In* and *Out* data set as built-in variables.

Except for the two logical operators, the operators in Table 1 can be applied both to integers and *Set*-valued variables. Using the +, −, and * operators generate code that perform Or, Minus, and And operations on bit vectors.

In dataflow analysis, one often needs to iterate over a subset of objects, so we added a *foreach* statement to do this. *Foreach* is a predicated iterator, meaning that it steps through the members of a set and performs actions on only selected members of the set. The user does not have to declare an iterator specifically, just a variable of the type over which the iteration is occurring and the set on which to iterate. The user may also specify a condition that acts as a filter and a direction (Forward/increase or Backward/decrease). The condition is described with the *where* keyword. The syntax is shown in Table 1.

The *type*, *range* and *condition* allowed are listed in the attached syntax table. The "where *condition*" and "*direction*" parameters are optional.

data types	Set Map int bool void
special variables	In Out
operators	$+ - * = += -= *=$ && $\|\|$
built-in classes	Opnd Instr Block Alias Expr Func Region
special methods	Init Compose Meet Result
built-in functions	DstAliasTable SrcAliasTable Print
built-in constants	Forward Backward
declarations	Phase *identifier* (*parameter list*) { ... }
	extend class *type* { ... }
	type TransFunc (*direction*) { ... }
statements	*lvalue* = *expression*;
	if (*expression*) { ... } else { ... }
	/% *arbitrary C++ code* %/
	foreach (*type var* in *range* where *cond. direction*) { ... }
	phoenix-iterator (...) { ... }

Table 1
AG Syntax Summary

Such *foreach* statements are translated to conditional for loops in the C++ and use the iterator macros in the Phoenix framework. Note that the *foreach* statement, especially the predication, is not strictly necessary (an additional *if* is sufficient), but the same can be said of C's *for* statement.

If the *range* is a Set, the *type* must match its content. Otherwise, if the *range* is a class, the *type* must match one of its members. For example, each instruction contains a list of operands, so we can specify a *type* of Opnd and a *range* of an instruction. Also, the user may specify a *condition* of "dataflow && dst" to iterate over dataflow-related destination operands in the list.

Phoenix provides a number of iterator macros, which can be used in AG almost verbatim (see Figure 3 Line 12). The only difference is that in C++, a matching "next" macro must follow the use of each iterator macro (see Figure 4 Line 26); this is not necessary in AG.

DstAliasTable is a reserved function that takes an alias tag x as parameter and returns a set of destination operands whose alias-tag is x. Similarly, *SrcAliasTable* returns all source operands with the same alias-tag.

5 An Example

To illustrate AG, we present a complete example: the classical "reaching definitions" dataflow analysis. The complete AG source is in Figure 3.

This algorithm computes the sets of definitions that reach the entry and exit points of each basic block in a program. Following the Dragon book [1], a definition of a variable is the operand in an instruction that may assign to the variable. In the Phoenix IR, each instruction has source operands and destination operands. For reaching definitions, we are concerned mostly with the destinations.

The whole analysis is defined as a phase called *ReachingDefs* (line 1 of Figure 3). The rest of the analysis consists of extend classes that add fields and computations to the built-in data types for operands, instructions, and basic blocks, and description of transfer functions.

5.1 Extend Classes

Extend classes augment existing data types with additional fields in which to collect information and procedures for collecting it. This is similar to extending a base class in an object-oriented language, but differs because the new attributes are actually attached to objects of the "base class" itself at the language level, not just in objects of derived classes (the C++ code we generate from AG actually uses class inheritance). But a user can refer to new attributes as if they were already in the original class. Consider the Opnd extend class (lines 3–20). This adds two attributes to each operand, operand sets named *Gen* and *Kill*. As usual, the *Gen* set contains operands that are defined within the block and available immediately after it in the source code.

The *Init* function initializes the values of the *Gen* and *Kill* fields. The two sets are implemented as bit vectors—see Lines 2–12 in Figure 4 for the declaration of *Gen*; Lines 14–29 show the translation of the *Init* function. The body of *Init* adds destination operands to the *Gen* set. Similarly, all other destination operands in the built-in destination-opnd-map-to-alias-tag table (DstAliasTable) that have the same alias tag as the operand (i.e., when both modify the same memory location) are added to the *Kill* set (Lines 12–17).

The *Instr* and *Block* extend classes add *Gen* and *Kill* sets to each of their classes and populate these sets with data from *Opnd* and *Instr* objects respectively. Lines 47–72 in Figure 4 call the three *Init* functions (the translation of the other two are not shown). Note that this function is synthesized completely from how this data is used in the analyzer, not from explicit code in the AG source.

After collecting *Gen* and *Kill* sets for blocks, the algorithm specifies some details of the main analysis iteration. At the beginning of the transfer function *TransFunc*, the iteration is declared to proceed in the forward direction and return a set of *Opnd* objects.

```
Phase ReachingDefs {

  extend class Opnd {
    Set<Opnd> Gen;
    Set<Opnd> Kill;

    void Init() {
      Opnd opnd = this;
      if (opnd->IsDef) {
        opnd->Gen += opnd;

        foreach_must_total_alias_of_tag(alias_tag, opnd->AliasTag, AliasInfo) {
          opnd->Kill += DstAliasTable(alias_tag);
        }
        opnd->Kill -= opnd;
      }
    }
  }

  extend class Instr {
    Set<Opnd> Gen;
    Set<Opnd> Kill;

    void Init() {
      Instr instr = this;

      foreach (Opnd dstOpnd in instr where (dataflow && dst)) {
        instr->Gen += dstOpnd->Gen;
        instr->Kill += dstOpnd->Kill;
      }
    }
  }

  extend class Block {
    Set<Opnd> Gen;
    Set<Opnd> Kill;

    void Init() {
      Block block = this;

      foreach (Instr instr in block) {
        block->Gen = instr->Gen + (block->Gen - instr->Kill);
        block->Kill = block->Kill + instr->Kill - instr->Gen;
      }
    }
  }

  Set<Opnd> TransFunc(Forward) {
    Compose(N) {
      Out = In - N->Kill + N->Gen;
    }

    Meet(P) {
      In += P->Out;
    }
  }
}
```

Fig. 3. A Complete AG analysis: Reaching Definitions

```
class OpndExtensionObject :
   public Phx::RbagGenTest::AG::OpndExtensionObject
{
  PHX_DECLARE_PROPERTY(Phx::BitVector::Sparse *, Gen);
  __PHX_DEFINED_VIRTUAL_GET_PROPERTY(Phx::BitVector::Sparse *, Gen) __const;
  __PHX_DEFINED_VIRTUAL_SET_PROPERTY(Phx::BitVector::Sparse *, Gen);

  Phx::BitVector::Sparse * _local_Gen;
}

void OpndExtensionObject::Init( Phx::FuncUnit *func_unit,
                                Phx::BitVector::Sparse *PHX_ARRAY(dst_alias_table))
{
  Phx::IR::Opnd *opnd = _this;
  if(opnd->IsDef) {
    this->Gen->SetBit(this->uid);
    foreach_must_total_alias_of_tag(alias_tag, opnd->AliasTag, func_unit->AliasInfo) {
      this->Kill->Or(dst_alias_table(alias_tag));
    }
    next_must_total_alias_of_tag;
    this->Kill->ClearBit(this->uid);
  }
}

void IterateData::Merge(
  Phx::DataFlow::Data *dependent_block_data,
  Phx::DataFlow::Data *effected_block_data,
  Phx::DataFlow::MergeFlags flags) {
  IterateData * dep_block_data = PTR_CAST(IterateData *, dependent_block_data);
  Phx::BitVector::Sparse * Out = dep_block_data->Out;

  if(flags & Phx::DataFlow::MergeFlags::First) In = Out->Copy();
  else In->Or(Out);
  dep_block_data->Out = Out;
}

void Traverser::InitData(Phx::BitVector::Sparse *PHX_ARRAY(dst_alias_table))
{
  foreach_block_in_func(block, funcUnit) {
    foreach_instr_in_block(instr, block) {
      foreach_dataflow_dst_opnd(dstopnd, instr) {
        OpndExtensionObject *ext_dstopnd =
           OpndExtensionObject::GetExtensionObject(dstopnd);
        ext_dstopnd->Init(funcUnit, dst_alias_table);
      }
      next_dataflow_dst_opnd;
      InstrExtensionObject *ext_instr =
         InstrExtensionObject::GetExtensionObject(instr);
      ext_instr->Init(funcUnit->Lifetime);
    }
    next_instr_in_block;
    BlockExtensionObject *ext_block =
        BlockExtensionObject::GetExtensionObject(block);
    ext_block->Init(funcUnit->Lifetime);
  }
  next_block_in_func;
}
```

Fig. 4. Part of the Phoenix (C++) code generated by the AG compiler for the reaching definitions example

The extend classes are based on original IR classes. The example in Figure 3 shows that the user may refer to fields from the extend class (e.g., Figure 3, Line 10, "`opnd->Gen`") using the same notation as for those in the base class (e.g., Figure 3, Line 13: "`opnd->AliasTag`"). These two references generate very different C++ code (c.f. Figure 4, Lines 21 and 23).

5.2 Transfer Function

As usual, we assume there are unique entry and exit points in the control flow graph for each block. "In" and "Out" are two built-in data sets related to the entry and the exit points respectively. The definition for TransFunc head declares the type of "In" and "Out" sets as holding operands. These two sets are usually used in the transfer function to pass data.

Compose and Meet are the two main functions for defining the transfer function. In this program, they specify the two groups of dataflow equations in the standard way [1, Eq. 10.9]:

$$in[B_i] = \bigcup_{B_j \text{ a predecessor of } B_i} out[B_j]$$

$$out[B_i] = gen[B_i] \cup (in[B_i] - kill[B_i]).$$

The first equation is exactly and simply included in the Meet function (Line 59), which computes the effect of the exit-point data from predecessors to the entry-point data of the current block in the iteration. *In* is related to the current block being visited, while *Out* is related to the block P that is passed to the *Meet* function. By default, the argument for the *Meet* function is a basic block that represents an arbitrary predecessor of the current block. As shown in Figure 4 lines 31–45, the data equation is translated into bit-vector manipulations.

The second dataflow equation is included in the *Compose* function (Line 55), which computes the data transformation globally from the entry point to the exit point for a single block. Declared as an argument to the *Compose* function, variable N is an extended object of the block by default. Since *Gen* and *Kill* are fields that have been added to the Block class (lines 38 and 39), they can be referred to as members of N.

5.3 Wrap up: Phase and Traverser

A complete AG program is translated into a C++ program that is compiled as a plug-in phase that can be invoked as part of the Phoenix compilation processes. It initializes all extended objects first, then executes the forward traverser, which applies the dataflow equations to iteratively compute on the blocks following the structure of the control-flow graph until the *In* sets converge for every block. The generated code uses the machinery built into the Phoenix framework to do this; an AG user does not write code for this.

	Reaching Definitions	Live Variables	Uninitialized Variables
C++ LOC (manual)	791	303*	108†
AG LOC (manual)	64	55	94
C++ LOC (generated)	626	519	682
C++ runtime	7.3s	0.8s	†
AG runtime	7.4s	3.1s	13.6s

*The manually-coded live variable analysis uses hard-coded fields, which makes it simpler at the expense of being far less modular.

†The manually-coded uninitialized variables analysis relies on the Phoenix SSA library not included in this count. This is a very different architecture than the code generated by AG.

Table 2
Experimental results: size and speed of AG-generated code vs. handwritten.

6 Experimental Results

We tested AG on three analyses: reaching definitions, live variables, and uninitialized variables. We chose these three examples because a hand-written version of each, done by experienced programmers, already existed in Phoenix. We compared the size and speed of the generated code with the manually written version for the first two examples because, like our generated code, they use the Traverser class in Phoenix. The manually-written version of uninitialized variables used Phoenix's static single-assignment code, which AG does not take advantage of, so we did not experiment with it.

Table 2 shows our results. "LOC" indicates the number of lines of code excluding comments; times are in seconds. We computed the average run times of these plug-ins by running compiler with the plug-in, running the compiler without the plug-in, and subtracting these two running times. The times are thus a little suspect because they also include the time to load and initialize the plug-in itself.

In each test case, the C++ code generated by the AG compiler is more than six times the size of the AG source. Even better for AG, the manually-written code for reaching definitions is even larger than the generated code. That is because the AG library files include commonly used code and default methods, for example, the constructor of the phase.

The manually-written live-variables code is smaller than the generated C++ code for that analysis, but this is because the manually-written code does not use the (verbose) Phoenix extend objects.

We ran the generated Phoenix C++ code on a laptop with a 2.0 GHz Pentium-M processor running Windows XP. The benchmark is the Phoenix

Microsoft Intermediate Language reader, which can generate high-level intermediate representations for a variety of targets. It is about five hundred thousand lines of code.

The AG-generated code for the reaching definitions analysis runs just as fast as the manually-written code on the MSIL reader. Unfortunately, the live variable analysis code runs about one-fourth as quickly, but there is a good reason for this: the manually-written C++ version does not use the Phoenix object-extension facility. Instead, it simply recomputes the desired data every time it traverses a block. Thus, the speed difference here more illustrates the cost of using extension objects instead a more brute-force approach. Evidently in this example, the computation is cheap enough so that repeating it is less costly than saving and recovering it later. We include the runtime for the AG code for uninitialized variables, but do not give a time for the manually-written code because it uses a completely different algorithm.

7 Conclusions

We presented a domain-specific language, AG, for writing dataflow analysis phases in Microsoft's Phoenix framework. Experimental results show that manually-written AG code can be less than one-tenth the size of the equivalent manually-written C++ with similar performance. A key enabler for the simplicity of AG code is its mechanism for extending existing IR classes, which makes it possible to extend existing classes without recompiling them and allows user-level code to access these fields as easily as typical ones.

As a small, domain-specific language, AG has some weaknesses. Minimizing verbosity was our focus, and we did so at the loss of some flexibility. The most obvious is that the user is forced to use the iterative analysis framework, even though Phoenix has other options, such as lattice and static single-assignment frameworks. Although AG has some high-level types such as sets and maps, its type system is limited and does not support strings, arrays, arbitrary iterators, and so forth.

AG is also currently limited to analyses running on the medium-level intermediate representation (MIR), although it could be extended to handle others. Furthermore, AG programs currently only handle user-defined variables; the many implicit temporary variables in the MIR are currently ignored. For example, the C statement on the left is dismantled as shown on the right. AG code currently ignores the temporary `t1`.

```
x = y + 3;        ⟶        t1 = y + 3;
                           x = t1;
```

As with many domain-specific languages, debugging AG is somewhat problematic. While we provide a print statement, AG does not have a dedicated debugger, IDE, or any of the other now-standard features in a development environment. All these could be added, but not without a fair amount of work.

AG is constructed as a translator, so in theory most weaknesses could be fixed by extending AG, provided the new features were supported by Phoenix. It could be extended, say, to describe region-based dataflow analyses, or to describe optimizations. But it is difficult to say at what point AG would cease to be a domain-specific language and balloon into C++.

Nevertheless, we believe that a factor of ten in code-size reduction justifies the extra challenges in using a small language.

For more information about Phoenix, see its official website: http://research.microsoft.com/compilers.

Acknowledgments

We would like to thank Al Aho for his enlightening discussions and interesting suggestions on this project. We also wish to thank the whole Phoenix group for their kind help and support.

References

[1] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers, principles, techniques, and tools," Addison-Wesley, 1988.

[2] Alt, M. and F. Martin, *Generation of efficient interprocedural analyzers with PAG*, in: *Proceedings of the Second International Symposium on Static Analysis (SAS)* (1995), pp. 33–50.

[3] Clifton, C., G. T. Leavens, C. Chambers and T. Millstein, *MultiJava: Modular open classes and symmetric multiple dispatch for Java*, in: *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Minneapolis, Minnesota, 2000, pp. 130–145.

[4] Dwyer, M. B. and L. A. Clarke, *A flexible architecture for building data flow analyzers*, in: *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, Berlin, Germany, 1996, pp. 554–564.

[5] Kam, J. B. and J. D. Ullman, *Global data flow analysis and iterative algorithms*, Journal of the ACM **23** (1976), pp. 158–171.

[6] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science **2072** (2001), pp. 327–355.

[7] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming* (1997), pp. 220–242.

[8] Kildall, G., *A unified approach to global program optimization*, in: *Proceedings of Principles of Programming Languages*, 1973, pp. 194–206.

[9] Nielson, H. R. and F. Nielson, *Bounded fixed point iteration*, in: *Proceedings of Principles of Programming Languages (POPL)*, Albuquerque, New Mexico, 1992, pp. 71–82.

[10] Tjiang, S. W. K. and J. L. Hennessy, *Sharlit: a tool for building optimizers*, in: *Proceedings of Programming Language Design and Implementation (PLDI)*, New York, New York, 1992, pp. 82–93.

[11] Venkatesh, G. A. and C. N. Fischer, *Spare: A development environment for program analysis algorithms*, IEEE Transactions on Software Engineering **18** (1992), pp. 304–318.

[12] Wilhelm, R., *Program analysis—a toolmaker's perspective*, ACM Computing Surveys **28** (1996), p. 177.

[13] Wilson, R. P., R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam and J. L. Hennessy, *SUIF: An infrastructure for research on parallelizing and optimizing compilers*, ACM SIGPLAN Notices **29** (1994), pp. 31–37.

[14] Yi, K. and W. L. Harrison III, *Automatic generation and management of interprocedural program analyses*, in: *Proceedings of Principles of Programming Languages (POPL)*, Charleston, South Carolina, 1993, pp. 246–259.

AG Syntax

ag-phase:
  `Phase` *identifier* `(` *parameter-list*$_{opt}$ `)` *compound-statement*

parameter-list:
  parameter
  parameter-list `,` *parameter*

parameter:
  type identifier

type:
  basic-type
  extensible-class-type
  `Set <` *type* `>`
  `Map <` *type* `,` *type* `>`

basic-type: one of
  `int  bool  void`

extensible-class-type: one of
  `Alias  Opnd  Instr  Block  Region  Func`

compound-statement:
  `{` *statements* `}`

statements:
  statement
  statements statement

statement:
  variable-declaration
  function-definition
  extend-class-definition
  assignment-expression$_{opt}$ `;`
  if-else-statement
  foreach-statement
  phoenix-foreach
  `continue ;`
  `break ;`
  `return` *expression*$_{opt}$ `;`
  cpp-code-segment
  compound-statement

variable-declaration:
    type variable-declaration-list `;`

variable-declaration-list:
    variable
    variable-declaration-list `,` *variable*

variable:
    identifier
    identifier `=` *expression*

function-definition:
    basic-function-definition
    transfer-function-definition
    compute-function-definition

basic-function-definition:
    type identifier `(` *parameter-list*$_{opt}$ `)` *compound-statement*

transfer-function-definition:
    type `TransFunc` `(` *direction* `)` *compound-statement*

compute-function-definition:
    compute-function-name `(` *identifier*$_{opt}$ `)` *compound-statement*

compute-function-name: one of
    `compose  meet  result`

extend-class-definition:
    `extend class` *extensible-class-type compound-statement*

assignment-expression:
    variable-or-field assignment-operator expression
    expression

variable-or-field:
    variable-or-field `->` *identifier*
    identifier

expression:
 numeric-literal
 variable-or-field
 expression binary-operator expression
 `!` *expression*
 `-` *expression*
 variable-or-field `(` *variable-list*$_{opt}$ `)`
 `(` *expression* `)`

variable-list:
 variable-or-field
 variable-list `,` *variable-or-field*

binary-operator: one of
 `+  -  *  <  >  &&  ||  <=  >=  !=  ==`

assignment-operator: one of
 `=  +=  -=  *=`

if-else-statement:
 `if (` *expression* `)` *statement*
 `if (` *expression* `)` *statement* `else` *statement*

foreach-statement:
 `foreach (` *type identifier* `in` *expression* `where`$_{opt}$ *direction*$_{opt}$ `)` *compound-statement*

where:
 `where` *expression*

direction: one of
 `forward  backward`

phoenix-foreach:
 phoenix-foreach-keyword `(` *parameter-list*$_{opt}$ `)` *compound-statement*

cpp-code-segment:
 `/%` *C++-program-text* `%/`

Automated Derivation of Translators From Annotated Grammars

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

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

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

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

Abstract

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

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

1 Introduction

1.1 Motivation

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

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

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

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

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

1.2 Approach

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

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

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

In summary, the main contributions of our technique are :

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

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

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

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

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

2 Context

2.1 Operations languages

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

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

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

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

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

2.2 The research problem

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

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

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

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

2.3 Our solution in a nutshell

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

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

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

3 Annotated Grammars

Syntax-directed translation [1] is a common mechanism used, mainly in compiler construction, to translate from a source to a target language. A particular instantiation of this technique is the use of syntax-directed transduction [15] that specifies the input-output relation of the translation and deduces the actual translator from that relation.

Our approach builds on these techniques to develop a simple and easy-to-use mechanism to *semi-automatically* build source-code translators between two related languages, taking as input the grammars of both languages, previously annotated with constructor and label information to establish a mapping [16] between corresponding language constructs. The mechanism provides a way to automatically generate the translator for some of the productions in the grammars, as well as basic support to extend that translator with the necessary transformations for the remaining productions.

Although many existing tools could be used to implement this solution, as for instance [3,7,9,23], we have chosen Asf+Sdf and the Asf+Sdf Meta-Environment for implementing our prototype. Asf+Sdf is a specification formalism composed of the Algebraic Specification Formalism (ASF) and the Syntax Definition Formalism (SDF), allowing the integrated definition of syntax and semantics of a programming language [6] in a modular way.

The modularity of Asf+Sdf enables reusability, at the syntactic as well as at the semantic level, which is one of the advantages of using it as our implementation medium. Furthermore, Asf+Sdf has a strong notion of syntax-directed translation both on input and output sides. We discuss SDF in more detail in Section 3.1, followed by a brief summary of ASF in Section 3.2.

3.1 SDF

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

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

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

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

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

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

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

3.2 ASF

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

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

stating that whenever L is matched, it can be rewritten to R, on the condition that $C_1, \ldots, C_n$ all evaluate to true. A simple form of equation is the unconditional one $L = R$. In the left-hand side, right-hand side and conditions of an equation, variables can be used. Matching a left-hand side of an equation implies binding of the variables to the matched subterms in the concrete syntax tree. See [6] for a more detailed description.

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

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

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

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

Fig. 3. Conditional constructs in two different languages.

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

3.3 Preliminary Experiment

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

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

3.4 Grammar Annotations

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

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

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

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

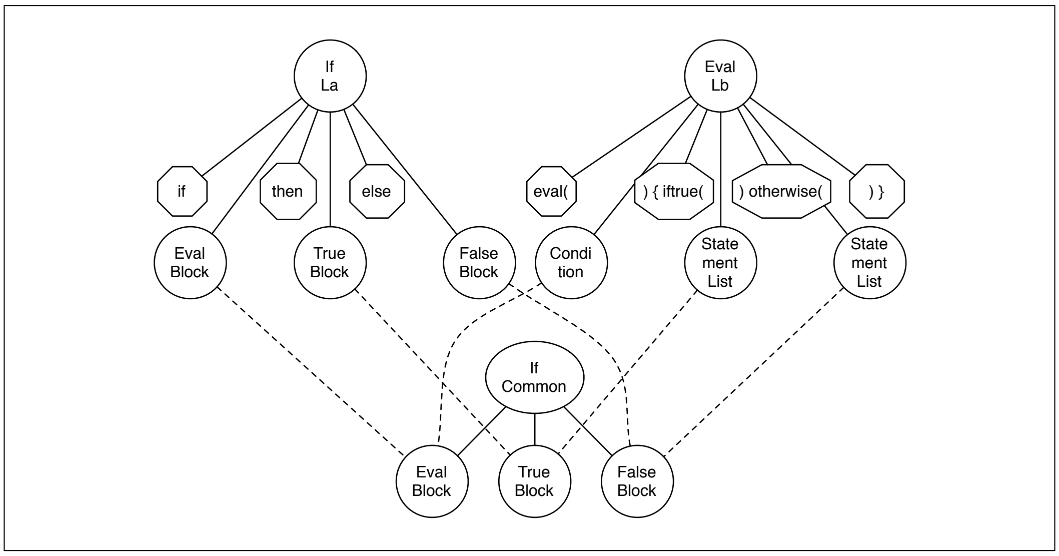


Fig. 5. Abstraction of a common construct.

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

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

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

As Figure 5 illustrates, such an equivalence can be regarded as an AST shared by the corresponding constructs in both languages. Since terminals (denoted by octagons in the figure) do not interest us when defining this correspondence, they are left out of the common AST. Like this we build a bridge between the two languages, allowing us to translate specific instances of a construct in one language to its counterpart in the other language. Figure 6 shows the SDF productions for the language constructs of Figure 3.

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

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

Fig. 7. Two annotated equivalent productions

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

3.5 Summary of the approach

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

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

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

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

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

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

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

3.6 Transformation Example

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

```
module Source
imports Expr
context-free syntax
"proc" b:StatsS "endproc"          -> StartS {cons("Start")}
"if" e:Expr "then" b:StatsS "fi"    -> IfS {cons("IfThen")}
"if" Expr "then" StatsS
    "else" StatsS "fi"              -> IfS {cons("IfThenE")}
"while" e:Expr "do" b:StatsS "od"   -> WhileS {cons("While")}
if:IfS | w:WhileS | e:Expr          -> StatS {cons("Stm")}
it:StatS*                           -> StatsS {cons("Block")}
```

Fig. 8. Part of the source grammar

```
module Target
imports Expr
context-free syntax
"start(" b:BlockT ")"                -> StartT {cons("Start")}
"eval(" e:Expr "," b:BlockT ")"      -> EvalT {cons("IfThen")}
"loop(" e:Expr "," b:BlockT ")"      -> LoopT {cons("While")}
if:EvalT | w:LoopT | e:Expr          -> InstT {cons("Stm")}
it:InstT*                            -> BlockT {cons("Block")}
```

Fig. 9. Part of the target grammar

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

Fig. 10. Part of the common grammar

The transformation system starts by relating productions in the source and target grammars with the same constructor attribute. The non-terminal at the right-hand side of the production in the source grammar becomes the argument of a translation function f, while the right-hand side of the production in the target grammar becomes the result of that translation function. E.g., for the productions with constructor attribute `cons("IfThen")`, a translation function `f(IfS) -> EvalT` will be derived.

The rewrite equations for the transformation system will now be generated based on the left-hand sides of both productions. The translation function `f(IfS) -> EvalT` will be expressed like

`f(if $Expr$ then $StatsS$ fi) = eval( $Expr$ , f($StatsS$) )`

where every non-terminal *NT* has been replaced by a variable $*NT*$. For every non-terminal, the corresponding translation function is invoked, except for non-terminals like *Expr*, that thanks to languages similarities and environment modularization, are imported by both the input and output grammars — this common grammar is shown in Figure 10.

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

Fig. 11. Signature of translation functions

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

Fig. 12. Equations of translation functions

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

Fig. 13. Translation example

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

3.7 Example of Manual Intervention

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

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

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

(ii) And we add an equation to rewrite the pattern:
```
f(if $Expr$ then $StatsS$ else $StatsS2$ fi) =
eval( $Expr$ , f($StatsS$) ) eval( not $Expr$ , f($StatsS2$) )
```

After this manual intervention we have obtained a complete translator that can translate any program in *Source* to *Target*. For instance, the program in the left column of Figure 13 gets translated to the one on the right.

4 Intermediate Representation Language (IRL)

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

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

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

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

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

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

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

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

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

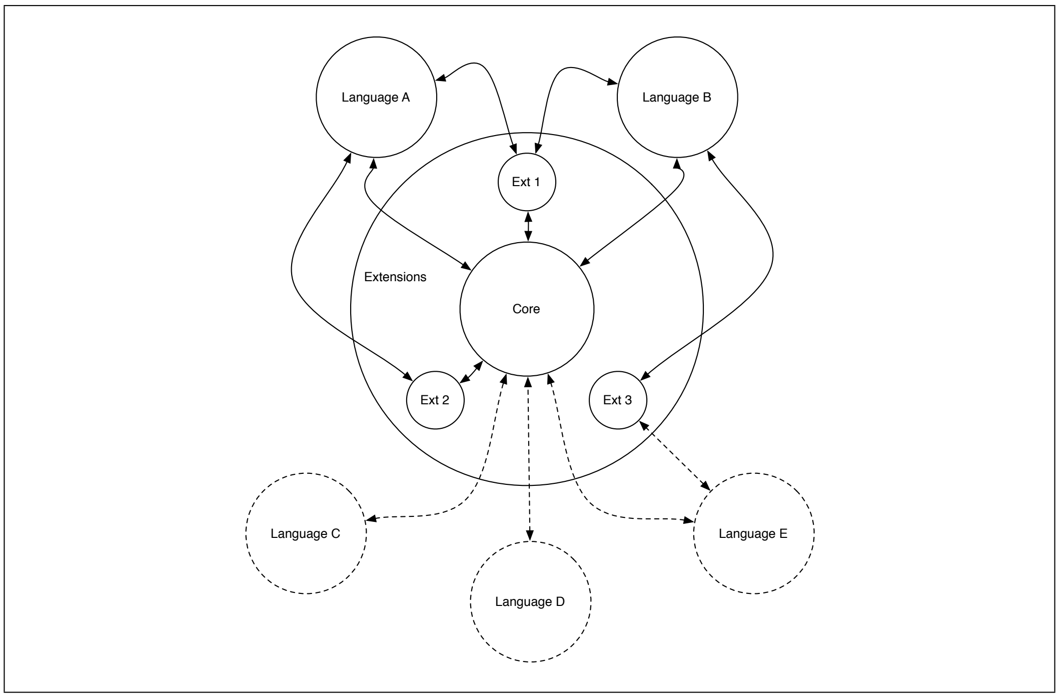


Fig. 15. Intermediate Representation Language structure

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

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

```
%% While statement

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

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

```
%% While

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

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

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

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

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

```
                %% Do - Until statement

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

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

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

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

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

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

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

5 Discussion

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

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

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

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

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

5.1 Related work

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

5.2 Future work

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

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

6 Conclusions

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

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

Acknowledgement

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

References

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

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

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

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

[5] Brand, M. v. d., A. v. Deursen, J. Heering, H. d. Jonge, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser and J. Visser, *The ASF+SDF Meta-Environment: a component-based language development environment*, in: R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, LNCS **2027** (2001), pp. 365–370.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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