Cost-Effective Maintenance Tools for Proprietary Languages

Merijn de Jonge
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Merijn.de.Jonge@cwi.nl

Ramin Monajemi
Lucent Technologies
Capitool 5, 7521 PL Enschede, The Netherlands
rmo@lucent.com

Abstract

Maintenance of proprietary languages and corresponding tooling is expensive. Postponing maintenance to reduce these costs is an often applied, short-term solution which eventually may lead to an unoperational toolset. This paper describes a case study carried out in cooperation with Lucent Technologies where maintenance cost is decreased by simplifying the development process of languages and tools. The development process is simplified by using a language-centered software engineering approach which increases software reuse and language dependent code generation. The case study was concerned with Lucent's proprietary SDL *dialect and involved the re-engineering of an* SDL *grammar and the construction of an* SDL *documentation generator.*

1. Introduction

Many companies in industry use proprietary programming languages or language dialects for the development of their software products. Consequently, such companies are faced with a maintenance effort for their software products *and* for the language and corresponding tooling. Maintenance cost of the language and tooling is usually high because a language change has great impact on the corresponding tooling [7].

To overcome such maintenance problems of language tools, a company has four options:

1. It might decide to stop the development of the language and tooling. This is a short term solution however, because knowledge of the tooling will gradually disappear which eventually may lead instead to an increase of maintenance cost.

2. Migrate its software products and implement them in a similar, international, standardized programming language (if available). After migration, commercially available language tooling can be used and the company can benefit from main stream language development. Of course, this approach can be very difficult, and may also have great impact on the development process of the software products and its developers, depending on the amount of discrepancies between the standard and the proprietary language.

3. Outsource the development and maintenance of its language tooling. This is a transfer of the problem to a third party, the maintenance problem still remains. Furthermore, it usually is extremely expensive and it makes the company dependent on a third party.

4. Decrease the maintenance cost by simplifying the development process of language tooling by increasing the amount of generated, language-specific code and by increasing the reuse of language-independent components.

In this paper we follow the last option. We describe a case study carried out in co-operation with Lucent Technologies in which we used a *language-centered* software engineering approach for the development of maintainable, inexpensive language tools for incremental documentation generation.

Language-centered software engineering is a component-based software engineering approach which assigns a central role to languages in the software development process. From language definitions a significant amount of code (including language-specific libraries and

components) is generated, which decreases maintenance effort after a language change by minimizing the required manual code adaptations.

This case study was concerned with Lucent's proprietary dialect of the Specification and Description Language (SDL), a language for the specification of the behavior of telecommunication systems. Lucent based its dialect on an early draft of the international standard, which only slightly differed from the final standard which appeared in 1988 [12]. To limit maintenance and development costs, Lucent Technologies decided long ago to 'freeze' their proprietary SDL dialect. Since adaptations to their SDL tool set were no longer required, knowledge of these tools disappeared gradually. This eventually led to increased development cost of new language tools and unavailability of existing tools due to old-fashioned hardware requirements that could no longer be met. Clearly, Lucent's early approach of reducing maintenance cost by freezing language and tool development has turned out to be counter-productive on the long term.

To demonstrate that language-centered software engineering simplifies language tool construction and that it can decrease maintenance cost, we started the development of a new tool environment for code browsing and visualization. This environment helps maintaining Lucent's SDL code because it improves accessibility of SDL code by providing access to the source code at different levels of abstractions and from different points of view. Since the environment can easily be extended, maintenance of SDL code can further be improved by connecting additional documentation and visualization components. Without preliminary knowledge about SDL, we developed a prototype tool environment in only a few man months time. After building some basic SDL components together, Lucent is now constructing its own language tooling using the techniques described in this paper.

The paper is organized as follows. We address language-centered software engineering in Section 2. That section motivates, amongst other things, the use of SDF for syntax definition. Section 3 discusses how an SDF grammar for SDL can be re-engineered from an operational YACC grammar. This SDF grammar is used in Section 4 where we demonstrate language-centered software engineering in practice, by developing an SDL documentation generator for code browsing and inspection. Sections 5 and 6 contain a discussion of related work, a summary of contributions, and directions for future work.

2. Language-centered software engineering

Grammars form an essential part of language tools, not only because they are used to derive parsers and pretty-printers from, but also because they greatly influence the shape of corresponding parse trees. Consequently, all parts of a software system that operate on parse trees depend on the grammar.

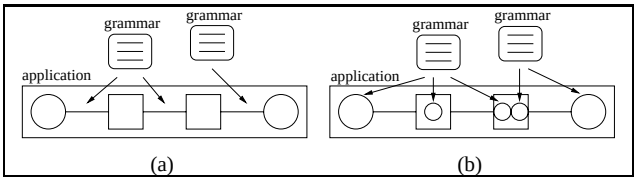

Figure 1. Grammars serve as contracts between different components (a) and for generation of language-specific code (b).

Different language tools perform several common subtasks, such as parsing, pretty-printing, and tree traversal. To optimize the software development process by minimizing code duplication, it is necessary that the code performing these tasks is shared between different language tools. Moreover, being free to choose a programming language that best suits the needs of an application is necessary as well. Sharing functionality between applications based on source code reuse only is therefore not sufficient. In addition to source code reuse, component-based software construction is required to share common tasks between applications written in different programming languages.

In [16] a language-centered software engineering approach is described that emphasizes the central role of grammars and the need for components in the software engineering process. It supports generation of stand-alone components and libraries from grammars as well as easy integration of off-the-shelf reusable components, and is based on the following key concepts.

Contracts. Building applications by connecting reusable, generated, and application-specific components requires agreement on the type and structure of data exchanged between these components. Further, a uniform exchange format is needed in order to connect such components easily. Since language tools typically transfer parse trees or abstract syntax trees between different components, the language itself describes the structure of the data that is transfered. In [16] an architecture of component-based software development is described where grammars serve as *contracts* between components (see Figure 1(a)).

Library generation. Since the structure of trees processed by language tools depends on the grammar, the code to traverse such trees also depends on the same grammar. Obviously, writing such code by hand is time consuming, error prone, and yields a maintenance problem because this code needs to be adapted over and over again whenever the

grammar changes. Moreover, this programming effort has to be repeated for each programming language in use. Generation of libraries containing tree access and traversal functions from the grammar for each programming language in use (the small circles in Figure 1(b)) therefore helps cost effective language tool development.

Component generation. In addition to the generation of programming language-specific libraries, the grammar can also be used to generate stand-alone language-specific components which can be reused as-is to construct new applications (the large circles in Figure 1(b)). Such components include parsers and pretty-printers, as well as tools to convert parse trees into abstract syntax trees and vice versa. See [16] for a discussion of the generation of language-specific components.

Language technology Language-centered software engineering requires state-of-the-art language technology for language definition and parsing. Reusability and maintainability of grammars are essential to fully benefit from language-centered software engineering. To meet these requirements, language technology is needed that:

- Accepts the full class of context-free grammars. This allows for clear encodings of languages, which do not have to be manipulated to fit in a restricted class of grammars.
- Offers a purely declarative syntax definition formalism. This prevents pollution of grammars with (application-specific) semantic actions that would hamper reuse and maintainability of grammars.
- Supports modular syntax definitions. Modularization allows language dialects to be defined as grammar extensions in separate modules. This prevents duplication of grammar definitions.

These requirements are fulfilled by the syntax definition formalism SDF [10, 28] together with generalized LR parsing [24, 28]. We used this technology in the case study presented here for SDL syntax definition and parsing.

Language-centered software engineering helps to decrease the development time of language tools because it minimizes the amount of language-specific code that has to be written by hand. Once a grammar exists, language-specific, stand-alone components and libraries are generated, which, together with off-the-shelf reusable components, help to build new language tools easily. Language-centered software engineering also improves software maintenance because due to code generation and component reuse, only a relative small part of an application requires manual adaptation after a language change.

Language-centered software engineering is supported by the tool bundle XT [15]. XT, which stands for 'Program Transformation Tools', bundles various related transformation tool packages into a single distribution. These packages include a generalized LR parser and parser generator [28], ATERMS [2] as uniform exchange format, the transformational programming language Stratego [29], and the generic pretty-printer GPP [13]. Furthermore, XT contains an extensive collection of grammar related tools and the Grammar Base [14], a collection of reusable grammars. XT is free software, requires minimal installation effort, and can be downloaded from `http://www.program-transformation.org/xt/`.

3. SDL grammar re-engineering

In Section 2 we discussed the requirements that language-centered software engineering puts on language technology and we motivated the use of SDF and generalized LR parse techniques. To benefit from these in order to build maintainable tools for SDL, we derived an SDF grammar from the EBNF definition and operational YACC grammar that were available for Lucent's SDL dialect. This section describes the systematic process that we followed to obtain a cleaned up grammar of this SDL dialect in SDF.

3.1. From YACC to SDF

Although SDF in combination with LR parsing offers advanced language technology which decreases maintenance costs of grammars and promotes their reuse, far more grammars have already been developed in YACC, many of which are operational and extensively tested. Since the development of a grammar from scratch is difficult and expensive, we propose systematic grammar re-engineering, in which an SDF grammar is (semi-automatically) obtained from an operational (legacy) YACC grammar, yielding a grammar that is as correct as the originating one.

We divided this grammar re-engineering process in four phases in order to make a clear distinction between different types of transformations (see Figure 2). During the first transformation phase, an SDF grammar is obtained from the YACC grammar. The remaining phases define source to source transformations on SDF in which a more natural encoding of the language is obtained. Since only the first two phases of the re-engineering process are YACC-specific and only the first phase actually depends on the YACC syntax, this re-engineering approach can also be defined for other syntax definition formalisms (like BNF and ANTLR [21]). Only a front-end which transforms a syntax definition to

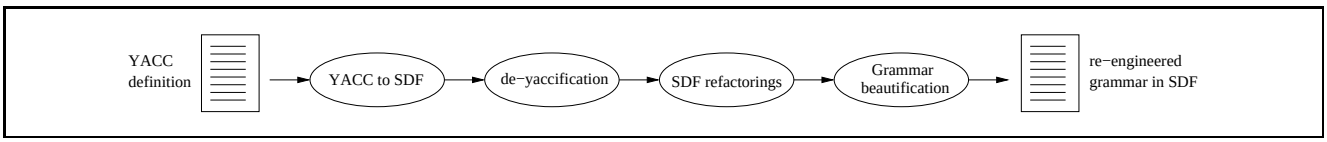

Figure 2. The four phases of the grammar re-engineering process in which an SDF definition is obtained from a (legacy) YACC definition.

SDF, and a transformer which removes system-specific constructs need to be defined.

YACC to SDF phase. The YACC to SDF phase is completely automated and basically consists of the following transformation:

$$\begin{array}{lcl} S & : & S_i \ldots S_{i+k} \\ & | & S_j \ldots S_{j+n} \\ & & \vdots \\ & ; & \end{array} \quad \Longrightarrow \quad \begin{array}{lcl} S_i \ldots S_{i+k} & \rightarrow & S \\ S_j \ldots S_{j+n} & \rightarrow & S \\ \multicolumn{3}{c}{\vdots} \end{array}$$

Observe that each alternative in the YACC definition yields a separate production in SDF, and that the productions are reversed with respect to formalisms like BNF. Furthermore, since SDF only allows syntax definition, all semantic related issues from the original YACC definition are removed during this phase. This includes removal of YACC's error handling mechanism using the reserved token "error".

Since we are interested in re-engineering grammars only, we ignore re-engineering semantic actions here. This holds for semantic actions that control the shape of parse trees as well as for semantic actions related to error recovery and error reporting. We use a generic parse tree format and consider modifying the shape of parse trees for particular needs as a separate phase *after* parsing. For error reporting and recovery we depend on generic mechanisms provided by our parser.

De-yaccification phase. An SDF grammar specification obtained in the previous phase is isomorphic to the grammar defined in YACC and therefore still contains YACC-specific idioms. These include lists expressed as left-recursive productions and productions introducing non-terminal symbols which only serve disambiguation (by encoding precedence and associativity in grammar productions).

In order to obtain a more natural specification of the language in SDF, we remove these YACC idioms, a process called de-yaccification [26]. We consider the following transformations:

- Introduction of ambiguous productions by flattening the productions that only serve disambiguation. This transformation also includes the addition of priority rules for disambiguation.
- Replacement of left-recursive list encodings by explicit list constructs.
- Joining lexical and context-free syntax by *unfolding* terminal symbols in context-free productions.

These transformations are performed automatically and have been implemented in the algebraic specification formalism ASF+SDF [1, 10, 6], a formalism supporting conditional rewrite rules based on concrete syntax. Its recent extension with traversal functions simplified the construction of transformation systems significantly. The system provides default bottom-up traversals and the programmer provides rewrite rules only for those constructs that need transformation. Since the individual de-yaccification transformations only affect a small part of the SDF grammar (which itself is quite large), only a few rewrite rules are needed to implement the transformations. Each transformation is defined as a separate ASF+SDF specification which can be executed in sequence to remove YACC-specific idioms step by step.

SDF refactoring phase. After the de-yaccification process, the grammar looks already more natural. The grammar still does not benefit from the powerful features of SDF however. The next step in the re-engineering process is to refactor the grammar to introduce such special SDF constructs which shorten the syntax definition and improve its readability. We consider the following transformations:

- Introduction of optionals.
- Introduction of SDF constructs for separated lists.
- Grammar decomposition by modularization.

We developed ASF+SDF specifications to automate the first two transformations. We do not have tool support for automatic modularization of grammars yet, but heuristics for grammar decomposition are described in [24, 27].

```
decision        : decisionStart decisionBody ENDDECISION
                | decisionStart error
                ;
decisionStart   : DECISION decisionValue endStmt
                | DECISION error
                ;
decisionBody    : caseList
                | caseList elseAnswer caseTransition
                ;
caseList        : case
                | caseList case
                ;
```

Figure 3. YACC fragment of SDL Decision construct.

Beautification phase. The previous transformations are general language-independent grammar transformations most of which can be automated. The transformations operate globally on the grammar and transform productions that match general patterns.

During the last phase of the re-engineering process, the grammar can be fine tuned by performing transformations operating on specific grammar productions. What transformations to perform largely depends on personal taste. An operator suite designed for expressing this kind of grammar transformations and applying them automatically is described in [17].

3.2. SDL grammar re-engineering

Lucent's proprietary SDL dialect is closely related to the SDL standard known as SDL 88 [12]. It was derived from an early (yet incomplete) draft of this standard. The absence of *block* constructs in the SDL dialect was the major difference with SDL 88. After SDL 88 many new language constructs have been defined in additional SDL standards of which SDL 2000 is the most recent one. Lucent's dialect did not benefit from these language progressions.

Both the syntax and semantics of Lucent's proprietary SDL dialect have been clearly defined in internal technical reports. Like standard SDL, the dialect exists in two forms: a graphical form (SDL-GR) and a textual form (SDL-PR). Since a mapping exists between both, we only considered the textual representation for the re-engineering project that we carried out. The lexical and context-free syntax of this language was defined in a single EBNF definition. In addition to the technical documentation we also had an operational YACC definition available which has been in use for years in the original SDL toolset. We considered this definition as ultimate starting point for re-engineering the SDL grammar. Unfortunately, the lexical analyzer that was also available turned out to be useless for this re-engineering process, because the lexical syntax was directly coded in C procedures and consequently hard, if not impossible, to re-engineer.

```
context-free syntax
  "DECISION" DecisionValue ";" DecisionBody
      "ENDDECISION" → Decision {cons("Decision")}

  Case+ ( ElseAnswer CaseTransition )?
      → DecisionBody {cons("DecisionBody")}
```

Figure 4. Re-engineered definition of the SDL Decision construct of Figure 3 in SDF.

The re-engineering process therefore consisted of a (semi-) automatic derivation of the context-free syntax from the YACC definition, and a manual definition of the lexical syntax. Due to the clear EBNF definition the mapping from the EBNF lexical syntax definition to SDF was straight forward (it was only a matter of minutes) and is not further addressed in this paper.

After the transformation to SDF was complete, we added constructor names to the context-free productions of the SDL grammar which are used to derive an abstract syntax definition (see Section 4). This annotated SDL grammar, allows tooling to operate on full parse trees (for example comment preserving pretty-printers), and on abstract syntax trees.

While testing the re-engineered grammar, we discovered that in existing SDL code slightly different lexical constructs were used as were defined in Lucent's EBNF definition. Since SDF definitions are modular, we were able to develop an extension to the SDL lexical syntax in a separate module to accept these constructs *without* affecting the lexical syntax definition that corresponds to the EBNF definition.

The re-engineering process transformed the YACC fragment of Figure 3 to the SDF fragment of Figure 4. Due to space limitations we omitted the semantic actions that were attached to most of the syntax productions in the original YACC definition. Observe that the re-engineered fragment contains fewer productions, that keywords are contained in the productions, and that SDF constructs such as optionals and lists are introduced.

The complete re-engineered SDL grammar is defined in 23 modules each defining particular SDL constructs. The number of non-terminals has been reduced from 254 to 104, the number of productions from 490 to 190.

4. An SDL documentation generator

This section addresses the development of an extensible documentation generator for SDL. The generator produces

Figure 5. Generated SDL documentation.

HTML documentation from SDL code. It uses extractors to collect specific information from SDL code which is used to provide different ways for code browsing. The documentation consists of a state name list and a pretty-printed SDL program. When clicking on a state name, the corresponding state definition is showed. A screen dump of generated documentation is depicted in Figure 5.

The documentation generator is based on the SDL grammar as developed in the previous section. From this grammar, we generate a parser which produces parse trees in the SDF parse tree format, called ASFIX [28]. ASFIX trees contain all information about parsed terms, including layout and comments. This enables the exact reconstruction of the original input term and the ability to process parse trees while preserving comments and layout. From the grammar also an abstract syntax definition can be derived, as well as pretty-print tables and Stratego signatures (see below).

Abstract syntax. From the concrete syntax definition, an abstract syntax definition can be derived based on the prefix constructor names defined as annotations of grammar productions (the cons attributes in Figure 4). These constructor names define the names of abstract syntax productions and can be added to the grammar manually or be synthesized automatically using the `sdfcons` tool. Abstract syntax trees can be derived from ASFIX parse trees because the constructor name information is contained in ASFIX trees. Language tooling can be developed to operate on parse trees or on abstract syntax trees depending on particular needs. The components of the documentation generator that we implemented operate on abstract syntax trees because they only require a subset of the information that is contained in the parse trees. The comment preserving pretty-printer that we reuse for pretty-printing SDL code on the other hand operates on parse trees.

Pretty-print tables. Parse trees and abstract syntax trees can be transformed into human-readable form by the generic pretty-printer GPP. This pretty-printer can produce several formats including plain text, LaTeX, and HTML. Precise, language-specific formattings are expressed in pretty-print tables which define mappings from language constructs (denoted by their constructor names) to BOX [3].

BOX is a language-independent markup language designed to define the intended formatting of text. Pretty-print tables can be generated from a grammar and customized to define the desired formatting. For example, a formatting for the Decision construct of Figure 4 can be specified as follows:

```
Decision --
 V[H["DECISION" _1 ";"] _2 "ENDDECISION"]
```

This formatting rule specifies that the keyword `DECISION`, the non-terminal DecisionValue and the semi-colon should be formatted horizontally. This horizontal box, together with the box representing the pretty-printed non-terminal DecisionBody, and the keyword `ENDDECISION` should be formatted vertically.

BOX supports labels in formatting rules which are translated to HTML anchors by the `box2html` formatter. With this labeling mechanism, we can add links to SDL language constructs in the generated HTML documentation. For the SDL documentation generator we are interested in state definitions such that we can jump in SDL code to the start of state definitions. Therefore, we use the LBL construct of BOX to define labels in the formatting rule of the State construct:

```
State --
 V[H["STATE" LBL["STATE" _1] ";"]
   _2 H["ENDSTATE" _3 ";"]]
```

Afterwards, an SDL-specific tool (`mk-labels`) replaces the label names as generated by the box generator (`STATE` in the example above) by unique names. Since the pretty-printer is language-independent it cannot be used here for this language-specific synthesis of label names.

After the generated pretty-print table for SDL has been customized to define appropriate formatting and labels for state definitions, generic tooling can be used to translate a parse tree to BOX and subsequently to one of the output formats. For the documentation generator we only use HTML, but the generation of documentation with richer formatting is supported via the `box2latex` back-end.

Stratego. We implemented SDL-specific components using the Stratego programming language. Stratego is a program transformation language based on term rewriting with strategies. It has an extensive library of strategies for term traversals and transformations. Stratego also supports the common exchange format ATERMs, which enables processing of parse trees and abstract syntax trees as produced

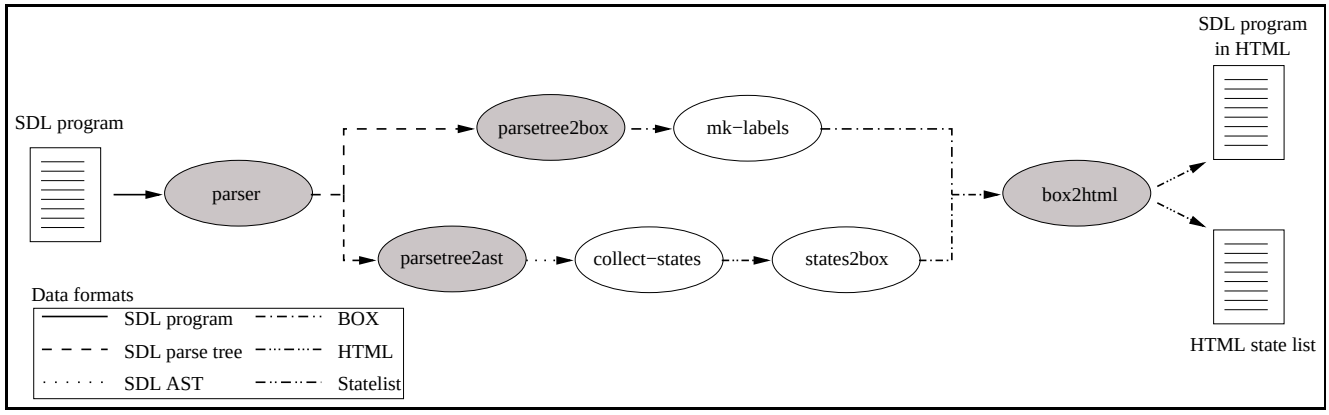

Figure 6. SDL documentation generation process.

by the SDL parser. In order to enable special term traversals and transformations, language-specific signatures are required by Stratego. These signatures which describe the shape of abstract syntax trees are generated from an SDF definition by the tool `sdf2sig`.

The SDL-specific `mk-labels` tool which inserts unique label names in a BOX term to denote the start of state definitions in SDL programs is implemented in Stratego as follows:

```
strategies
 mk-labels = topdown(try(mk-label))
 abox2str  = collect(mk-string); Hd
rules
 mk-string: S(str) -> str
 mk-label:
  LBL("\"STATE\"",abox) -> LBL(name, abox)
  where
   !["state:",<abox2str>abox];
   concat-strings => name
```

This program performs a top-down traversal of a BOX term, and tries to apply the rule `mk-label` to each node. This rule replaces the first argument of label nodes of the form `LBL("\"STATE\"", abox)` by the state name, preceded by the string `"state:"`. The state name is retrieved from the second argument of the `LBL` term. The program thus transforms a term `LBL("\"STATE\"",[S("my_state")])` into the term `LBL("state:my_state",[S("my_state")])`.

Once appropriate label names have been inserted by `mk-labels`, the BOX term can be translated to HTML using the `box2html` back-end. The resulting HTML pages contain anchors at the start of state definitions.

Extraction. The documentation generator generates a web-site which displays an SDL program to the user and allows him to browse this code in different ways. One way to browse the code is by offering a list of states names which, when clicked on, jump to the start of the corresponding state definition. To implement this, a code extractor needs to be implemented to collect a list of states from an SDL program. This is easily implemented in Stratego by traversing over the abstract syntax tree and collecting all nodes that correspond to state definitions:

```
strategies
 collect-states = collect(get-state-name)
rules
 get-state-name:
  State(StateList(names),_,_) -> <Hd>names
 get-state-name:
  State(StateList,_,_)        -> "*"
```

This program matches state nodes and distinguishes ordinary state definitions and default states (‘*’ states). For ordinary state definitions, the first of the list of state names is returned (with `<Hd>names`). For default states, `*` is returned as state name.

Collection of states is a simple extraction but when combined with other (more advanced) extractors, a rich SDL toolset can be constructed. A more complex example is the state transition extractor which is discussed later.

Connection to the documentation generator. To integrate the state collector in the documentation generator, its output should be represented in HTML as a list of hyper-links each pointing to the start of the corresponding state definition in a pretty-printed SDL file. These hyper-links should use the same label names as generated by the `mk-labels` tool described above. Following the language-centered software engineering approach, we consider transforming the output of the state collector to HTML as a separate step to be performed by a separate component. Therefore, the state collector does not produce HTML directly and, as a result, can also be used in different settings.

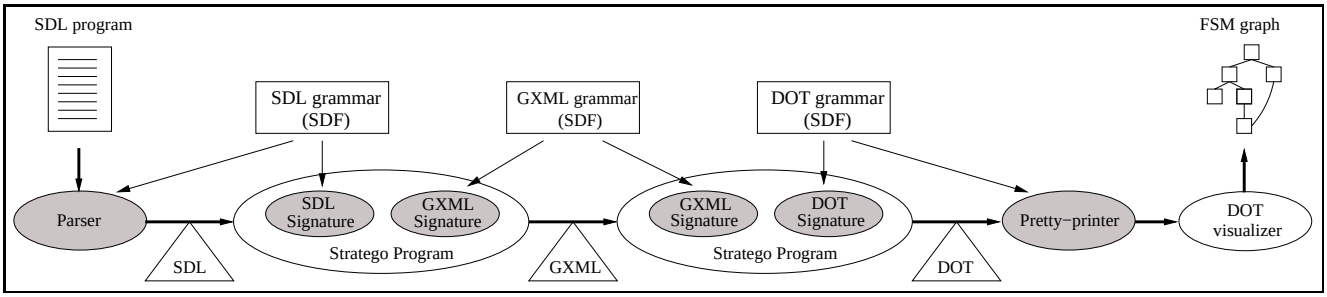

Figure 7. Architecture of the FSM graph generator.

To obtain an HTML representation of the list of states, we can construct a grammar for the output format and a pretty-print table to define the mapping to HTML. Because this approach has some overhead (it requires an additional grammar and extra parsing of the list of states), we followed a different approach and developed a small component that transforms the output directly to a BOX term (more precisely it produces an abstract syntax tree of a BOX term). This term can be passed to `box2html` to obtain the desired HTML representation.

Integration of components. All ingredients of an initial documentation generator have now been developed. We can format SDL programs in HTML by parsing a program and transforming the parse tree to BOX using the SDL pretty-print table. Then we can insert unique label names in the BOX term and transform it to HTML. The HTML list of state names (which link to state definitions) can be obtained by first parsing an SDL program and transforming the parse tree to an abstract syntax tree. Then we can collect all state names and translate the resulting list of states to BOX and finally to HTML. Figure 6 contains an overview of all components involved in this documentation generation process and shows how both the HTML state list and the pretty-printed SDL code are produced. Grey ellipses denote generated or reused components.

A FSM graph generator. The documentation generator is extendible and extra tooling can be developed to provide additional documentation and information of SDL programs. In addition to the state collector, we developed a finite state machine (FSM) graph generator. This generator produces, given an SDL model, the graph representation of the underlying FSM. In Figure 7 a detailed overview of the graph generator is depicted. Grey ellipses denote the components that are generated.

In addition to the SDL grammar, two more grammars are used for this tool. DOT [9] is a low level graph representation, which we used because off-the-shelf graph visualization tools for this representation were available for reuse. GRAPHXML [11] is a high-level graph representation language in XML in which a graph can be represented in terms of its mathematical description (i.e. in terms of edges and transitions). The grammars for DOT and GRAPHXML are available in the Grammar Base and reused here as off-the-shelf *language components*.

The only thing that needs to be implemented for the FSM visualizer is part of a single Stratego program (the left-most Stratego program in Figure 7), which is responsible for the extraction of the FSM information from SDL code and the generation of a graph representation in GRAPHXML. The tool that transforms GRAPHXML to DOT and the DOT visualizer were reused as-is. Both Stratego components share the generated grammar signatures. In Figure 8 a generated FSM graph is depicted that was extracted from a real-world SDL program of approximately 30.000 lines of code.

The FSM state generator is another example of an extractor and can easily be integrated in the SDL documentation generator. This is achieved by automatically converting the graph into a clickable image such that clicking on a node in the graph jumps to the corresponding state definition.

Maintainability Thanks to component reuse and code generation, the documentation generator could be implemented with little programming effort. All components together required less then 150 lines of Stratego code. Maintenance cost of these components is low because due to the generic term traversals of Stratego, language dependence of the components is limited, reducing the amount of code that needs to be adapted after a language change. For instance, the `collect-states` tool only depends on two SDL language constructs related to state definitions. Only when these constructs are changed in the language, the tool needs modification. Together with the modularization mechanism of SDF, this also greatly simplifies the simultaneously development of components for multiple SDL dialects. Additionally, component reuse and code generation also makes extending the generator relatively easy.

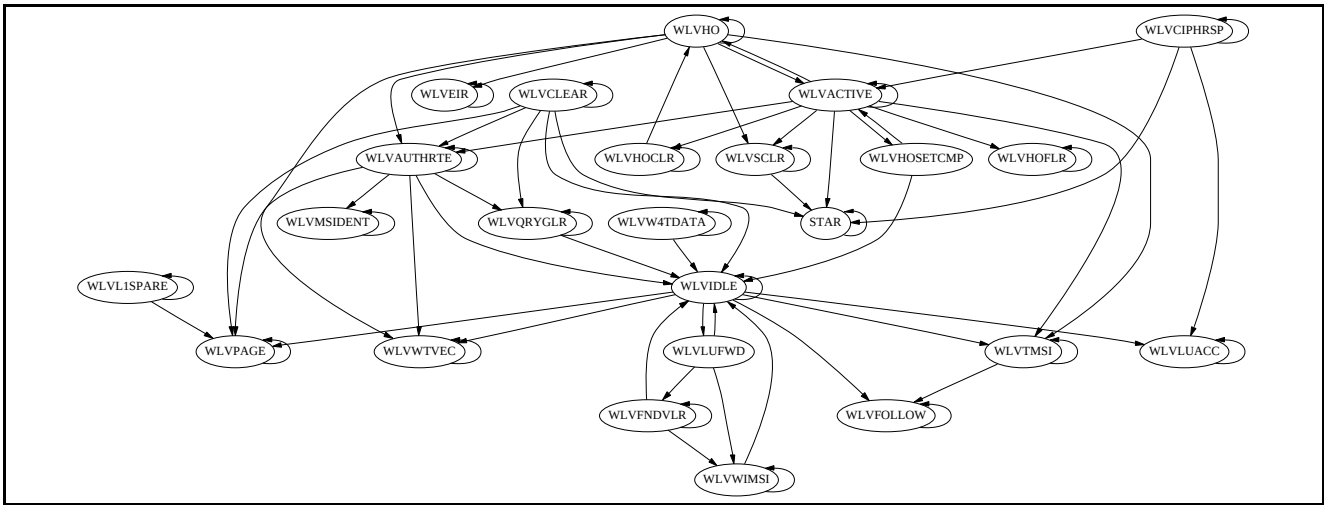

Figure 8. A generated FSM graph.

5. Related work

In [18], a semi-automated grammar recovery project is described where a complete grammar for VS COBOL II is constructed from an online manual. Grammar recovery from BNF definitions is discussed in [27]. In contrast to our approach based on an operational YACC definition, these approaches require grammar correction because they are based on non-operational language descriptions which often contain errors. A re-engineering approach similar to ours, not requiring grammar correction is described in [26]. They also derive an SDF grammar from an operational YACC definition but their approach yields grammars which are not optimal for software development. Heuristics for deyaccification are described in [31]. The authors focus on abstract syntax derivation from concrete syntax which benefits from a clear natural encoding of a language. Formalization of grammar transformations is addressed in [17]. They describe an operator suite for grammar adaptation which is derived from a few fundamental grammar transformations and supplemental notions like focus, constraint, and yielder.

In addition to the SDL grammar and bottom-up, generalized LR parser that we described here, the development of a *top-down* parser for SDL 2000 using ANTLR [21] as parser generator is described in [25]. Another approach using recursive descent parsing with backtracking is described in [5].

In addition to XT, many environments and tools exist for program transformation. The online survey of program transformation [30] strives to give a comprehensive overview of program transformation and transformation systems.

Hypertext for software documentation is discussed in several papers [4, 22, 20, 23]. Our SDL documentation generator was inspired by DOCGEN [8], a generator for interactive, hyperlinked documentation about legacy systems. They use Island Grammars (i.e. partial syntax definitions) for code extraction instead of full grammars as we do. A less precise extraction approach based on lexical analysis only is discussed in [19].

6. Concluding remarks

Contributions. This paper is concerned with grammar re-engineering and construction of maintenance tools for proprietary languages and dialects. The paper demonstrates that language-centered software engineering decreases development time of such language tools: once an SDF grammar for SDL was developed, implementing the tools described in this paper was only a matter of several man-hours. This is because language dependent components and libraries are generated and because existing (third party) components can be reused and integrated easily. Semi-automatic grammar re-engineering techniques brings language-centered software engineering into reach because it simplifies the move to essential state-of-the-art language technology significantly. We demonstrated that modular syntax definition, generation of language-specific code, and language independence of Stratego programs helps maintaining multiple language dialects. The techniques presented in this paper are currently being used within Lucent Technologies to further develop the SDL documentation generator and related tools.

Future work. As extension to abstract syntax derivation, we want to investigate DTD generation from concrete syntax definition (either from YACC or SDF definitions). Furthermore, we want to apply the techniques that we used for

the SDL dialect to other languages (including standard SDL and other proprietary SDL dialects), and extend the generator to provide all kinds of information of SDL programs in addition to state definitions and state transitions. Finally, we want to connect the documentation generator to existing re-engineering and documentation tools. Only simple modifications are required to connect existing tools through the ATERMs exchange format.

Acknowledgments. The authors thank Mark van den Brand, Arie van Deursen, Paul Klint, Ralf Lämmel, and Joost Visser for helpful comments on a draft of the paper.

References

[1] J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[2] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[3] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1 – 41, 1996.

[4] P. Brown. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3):363–392, 1991.

[5] P. Csurgay. Prototyping framework for SDL with evolving semantics. In J. Wu, S. T.Chanson, and Q. Gao, editors, *proceedings of the IFIP TC6 WG6.1 FORTE/PSTV'99*, Formal Methods for Protocol Engineering and Distributed Systems. Kluwer Academic Publishers, 1999.

[6] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[7] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[8] A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.

[9] E. R. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

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

[11] M. M. I. Herman. GraphXML – a graph description language. In *Proceedings of the Symposium on Graph Drawing (GD 2000)*, LNCS. Springer-Verlag, 2000.

[12] ITU-T recommendation Z.100, specification and description language (SDL), 1988.

[13] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[14] M. de Jonge, E. Visser, and J. Visser. The grammar base. Available from `http://www.program-transformation.org/gb/`.

[15] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. van den Brand and D. Parigot, editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[16] M. de Jonge and J. Visser. Grammars as contracts. In *Generative and Component-based Software Engineering 2000*, LNCS, Erfurt, Germany, To appear. Springer-Verlag.

[17] R. Lämmel. Grammar adaptation. In *Proceedings of Formal Methods Europe (FME)*, LNCS, Berlin, 2001. Springer-Verlag.

[18] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery, July 2000. Submitted, available at `http://www.cwi.nl/~ralf/`.

[19] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering Methodology*, 5(3):262–292, 1996.

[20] C. d. Oliveira Braga, A. von Staa, and J. C. S. do Prado Leite. Documentu: A flexible architecture for documentation production based on a reverse-engineering strategy. *Journal of Software Maintenance*, 10:279–303, 1998.

[21] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, July 1995.

[22] V. Rajlich. Incremental redocumentation with hypertext. In *1st Euromicro Working Conference on Software Maintenance and Reengineering CSMR 97*. IEEE Computer Society Press, 1997.

[23] V. Rajlich. Incremental redocumentation using the web. *IEEE Software*, 17(5):102–106, September/October 2000.

[24] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[25] M. Schmitt. The development of a parser for SDL-2000. Technical Report A-00-10, Medical University of Lübeck, 2000.

[26] M. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 245–255. IEEE Computer Society Press, 1999.

[27] M. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000.

[28] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[29] E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *LNCS*, pages 30 – 44. Springer-Verlag, 1999.

[30] E. Visser et al. The online survey of program transformation. `http://www.program-transformation.org/survey.html`.

[31] D. S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 472–480, Berlin - Heidelberg - New York, May 1997. Springer-Verlag.