Implementing the Front-End of an SDL Compiler

Marko Mäkelä

November 1998

Master's Thesis

Helsinki University of Technology
Department of Computer Science and Engineering
Theoretical Computer Science Laboratory

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä: Marko Mäkelä
Työn nimi: Implementing the Front-End of an SDL Compiler
Suomenkielinen nimi: SDL-kääntäjän etupään toteutus

Päivämäärä: 30. marraskuuta 1998 | **Sivumäärä:** 69

Osasto: Tietotekniikka
Professuuri: Tik-79 Digitaalitekniikka

Valvoja: Prof. Leo Ojala | **Ohjaaja:** TkT Nisse Husberg

Tiivistelmä:

Laajoissa hajautetuissa järjestelmissä esiintyviä virheitä on käytännössä mahdotonta löytää yksinkertaisen päättelemisen tai testaamisen avulla, ja virheillä voi olla odottamattoman kalliita seurauksia. Hankalimmatkin virheet paljastavien formaalien menetelmien käyttäminen on perinteisesti edellyttänyt tarkasteltavan järjestelmän mallintamista käsin. Monia järjestelmiä ei ole verifioitu, koska mallin tekemistä on pidetty liian työläänä.

Kääntäjät mullistivat tietojenkäsittelyn mahdollistamalla korkean tason ohjelmointikielet. Nyt kääntäjien tarkoituksenmukainen soveltaminen lisää formaalien menetelmien käytettävyyttä. Tämä työ kuvaa SDL-kääntäjän etupäätä, joka jäsentää ja tallentaa syntaksipuuhun SDL-kielisiä spesifikaatioita sekä suorittaa semanttisia tarkistuksia.

Jäsennin toteuttaa makroja lukuunottamatta koko SDL-kieliopin vuodelta 1996. Kieliopin saattaminen jäsennintyökalun edellyttämään LALR(1)-muotoon vaati lukuisia muutoksia kielioppiin ja selaimeen. Työssä kiinnitettiin erityistä huomiota virheiden sietämiseen, virheistä toipumiseen sekä virheilmoitusten havainnollisuuteen.

Valmistuttuaan kääntäjä muuntaa SDL-ohjelmia korkean tason Petri-verkkomalleiksi, joita voidaan verifioida formaalien menetelmien avulla.

Avainsanat:

leksikaalinen analyysi, jäsentäminen, syntaksipuu, semanttinen analyysi, kääntäminen, SDL, formaalit menetelmät, saavutettavuusanalyysi

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author: Marko Mäkelä
Name of the Thesis: Implementing the Front-End of an SDL Compiler
Name in Finnish: SDL-kääntäjän etupään toteutus

Date: November 30, 1998 | **Pages:** 69

Department: Computer Science and Engineering
Professorship: Tik-79 Digital Systems Science

Supervisor: Prof. Leo Ojala | **Instructor:** D.Sc. (Tech.) Nisse Husberg

Abstract:

Errors in large distributed systems are practically impossible to find with simple reasoning or testing, and they can have unexpected consequences that cost lots of money. Traditionally formal methods, which will find even the most obscure errors, have been based on manually derived models of the systems to be analyzed. Many industrial designs have not been formally verified, because the effort of modeling them has been considered too high.

Compilers made a revolution in computing by allowing the use of high-level programming languages. Now compiler technology can be applied to make formal methods more accessible for system designers. This work describes the front-end of an SDL compiler, which parses specifications written in SDL and stores them in a syntax tree and performs semantic checks.

With the exception of macros, the parser recognizes the whole SDL grammar defined in 1996. Converting the grammar to the LALR(1) form required by the parser generator tool has involved numerous modifications to the grammar and to the lexical analyzer. Special attention has been paid to error tolerance and recovery and to meaningful error reporting.

Once finished, the compiler will translate SDL specifications to high-level Petri Net models that can be verified using formal methods.

Keywords:

lexical analysis, parsing, syntax tree, semantic analysis, compiling, SDL, formal methods, reachability analysis

Acknowledgements

This Master of Science thesis has been prepared at the Theoretical Computer Science Laboratory (former Digital Systems Laboratory) of the Helsinki University of Technology. The work was done in the MARIA project financed by the Technology Development Centre of Finland (TEKES), Nokia Research Center, Nokia Telecommunications, Helsinki Telephone Corporation and Ratahallintokeskus, the Finnish railroad administration.

I wish to thank my supervisor, Professor Leo Ojala, and my instructor, D.Sc. Nisse Husberg, for their guidance during the work. My special thanks go to Eero Lassila and to others who proofread the thesis and provided me with some valuable comments and suggestions. I also use this opportunity to thank Matti Luukkainen of Nokia Research Center, Rick Reed of TSE Limited and Charles Lakos of University of Tasmania for helping me to understand some aspects of SDL better, and my colleagues at the Theoretical Computer Science Laboratory and at the Laboratory of Information Processing Science for creating a nice working atmosphere.

Marko Mäkelä
Otaniemi, November 30, 1998

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The rapid development in telecommunications and networking has made concurrent and distributed computer systems very common. Developing distributed systems is quite different from traditional sequential programming. Sequential programs always behave in the same way given the same input, but the operation of a concurrent system depends on a number of practically irreproducible conditions, such as the relative speeds of the system components, the congestion level in the network, or the exact global state of the system.

Less formal methods, such as visual inspection of program code, testing and debugging, are fairly efficient when applied to sequential programs, but computing systems exhibiting parallelism, especially distributed systems, require exhaustive testing. Without formal verification, one can never be sure that a system involving parallelism operates properly under all circumstances. Formal verification methods can of course also be applied to sequential computing systems.

Traditionally, there has been a large gap between the formalisms used for formally describing concurrent systems and the programming languages used for actually implementing them. In order to be able to formally verify that an implementation of a concurrent system works, one must know both the programming language and the formal description language. Manually translating the implementation into a formal description involves lots of work, and the process is very error-prone for large systems.

Automating the translation from an implementation into a system model suitable for formal verification will save much labor, reduce the possibilities for error and ease the use of formal verification tools. It will also hopefully make formal verification methods more attractive to the average system designer.

1.1.1 The MARIA Project

MARIA (Modular Reachability Analyzer) is an analyzer for concurrent and distributed systems that is being developed at the Theoretical Computer Science Laboratory (former Digital Systems Laboratory) of Helsinki University of Technology.

The analyzer uses a high-level Petri Net [3, 20] formalism. As its name hints, MARIA consists of modules: input modules (for supporting different classes of Petri Nets), method modules (for optimizing the analysis for different types of problems) and several other modules that implement necessary auxiliary functions.

The average engineer has never heard of Petri Nets. He is not likely to use any reachability analyzer if it cannot analyze systems written in the programming language he is used to. For this reason, MARIA must support languages known by the engineers. The support will be implemented in separate *language front-ends*, compilers that translate programs written in a source language into Petri Net models that can be processed by the analyzer. If an error state is found in the analysis, the results will be translated back into the source language, so that the engineer can see which statement sequence in his program leads to the error.

1.1.2 SDL

Telecommunication protocols are a very important class of distributed systems. Protocols are often specified in a language called SDL (CCITT Specification and Description Language) [27]. Because of the importance of telecommunications, the first language front-end implemented in the MARIA project will be for SDL.

The CCITT Specification and Description Language has evolved from a semi-formal graphical notation used for annotating the text in early CCITT Recommendations[1] to a language with formally defined semantics. Today SDL is not merely a specification language; there are code generators that translate SDL into programming languages such as C [4]. In fact, several companies use SDL (often extended with their own constructs) as an implementation language. Nokia Telecommunications uses its own language based on the 1988 version of SDL, called TNSDL (TeleNokia SDL). There is an analyzer for TNSDL called EMMA [17], and the Maria project was essentially prompted by the experiences from the EMMA project.

The ITU-T Recommendation defines two forms of SDL. The Graphical Representation (SDL/GR) consists of drawing symbols augmented with some grammar rules from the Phrase Representation (SDL/PR), which is a purely textual language. The graphical form of SDL is supported by some SDL tools. Until recently, ITU-T did not say anything about how SDL/GR specifications should be stored in files. [29] defines a Common Interchange Format for SDL/GR that is very similar to SDL/PR. Practically all SDL authoring tools can export definitions in SDL/PR format, and also SDL compilers work with SDL/PR.

SDL has been developed during a long period by a very heterogenous group of people. As a result of this, the language seems to contain many compromises, which are difficult for compiler writers and sometimes even not very intuitive for the user. But it is the most widely used language in the telecommunications sector. There are millions of lines of code written in SDL, sometimes combined with data type definitions written in ASN.1 [26, 28], which is another challenge for compiler writers and users new to the ITU-T way of thinking. Charles Lakos from

[1] ITU-T is the Telecommunication Standardization Sector of the International Telecommunication Union and was previously known as CCITT (Comiteé Consultatif Internationale de Télégraphique et Téléphonique).

the University of Tasmania recently got acquainted with SDL and formulated his thoughts in his review of [9], which is one of the best introductory books to SDL:

> SDL seems to have an excess of similar notions, such as systems, blocks, processes, partitioned blocks, services, etc. Sometimes it is unclear whether enhancements like block partitioning, channel partitioning, etc. are syntactic conventions for dealing with large models or whether they have semantic content as well. (It is suggested that this redundancy is a result of the committee approach to the definition of the language.)
>
> SDL seems to carry a lot of historical baggage, e.g. a specification can specify a singleton instance (and its type) or a type which can then be instantiated multiple times.
>
> SDL supports the definition of abstract data types (ADTs), referred to as "sorts." These do not have implementation constraints such as a fixed range for integers. It is not clear where the boundary is between the specification of ADTs and other forms of data such as signals.

SDL used to be a *case insensitive* language, meaning that it does not matter whether words are written using capital (upper case) or normal (lower case) letters. Combining *case sensitive* languages[2] with a case insensitive language leads to many problems. The SDL standardization bodies are considering making SDL case sensitive, and the Recommendation covering the combination of ASN.1 with SDL [28] will probably be largely rewritten before the next revision of SDL is published.

1.1.3 The SDL Front-End for MARIA

When finished, MARIA will analyze protocol specifications written in SDL. Since SDL is often combined with ASN.1, also ASN.1 will be supported. The SDL front-end consists of three major parts. First, it has *parsers* for SDL and ASN.1. Some *semantic analysis* is performed on the parsed input, and finally, a *model generator* translates the protocol specification into a high-level Petri Net model of the system written in a language understood by the reachability analyzer.

The work on MARIA is in a very early stage. It is not exactly clear what kind of Petri Nets the analyzer will support, and the format of the Petri Net description language has not been agreed upon yet. Because of this, so far only the first two stages of the SDL front-end have been written, namely the parsers for SDL and ASN.1, which include some semantic analysis.

This thesis concentrates on the strictly SDL-related parts of the front-end: the SDL parser, syntax tree generation and some semantic analysis for SDL. The future steps in this project are discussed in Section 7.2: integrating the ASN.1 parser to the front-end, and generating a high-level Petri Net model of the system.

[2] Practically every modern programming language is case sensitive.

The compiler is mostly based on the 1996 version of ITU-T Recommendation Z.100 [27]. Some shortcuts were taken. Most notably, data types cannot be defined axiomatically, and the compiler is case sensitive, which makes it much easier to implement support for ASN.1 definitions later.

1.2 Implementation Choices

In the beginning of a big software project, suitable tools must be chosen. Compilers are typically very complex programs, and bad implementation choices can make a compiler either perform very badly or awkward to extend later, or both.

1.2.1 Compiler Generation Tools: Flex and Bison

It seems that both SDL and ASN.1 have originally been designed by people who have little experience in implementing parsers. It is true that both languages [26, 27] are defined with a Backus–Naur Form (BNF) grammar, but it would be extremely naïve to assume that one could have an LALR(1)[3] parser generator transform the grammar to a working parser. The grammar must be adapted, as discussed in Chapter 2.

There may be commercially available or less known parser generation tools that support a larger class of grammars than LALR(1), but since the goal of the MARIA project is to develop a freely available, easily portable toolkit for reachability analysis, such tools were out of question.[4] We considered using the SDL parser developed at Humboldt-Universität zu Berlin but decided to write our own parser, because [16] requires a modified version of a term processor called Kimwitu, a program that seems to lack any support and is not very commonly known.

Traditional compiler construction tools in the C programming environment are Lex, the lexical analyzer generator, and Yacc (Yet Another Compiler Compiler, an LALR(1) parser generator), and their improved versions, Flex [18] and Bison [6]. In this project, we considered using an enhanced version of Berkeley Yacc, Backtracking Yacc [8], a tool for generating non-deterministic parsers, meaning that it can generate parsers for a larger class of grammars than traditional Yacc or Bison. Since there is always a risk involved when using a relatively unknown software component, we decided to play it safe and chose Bison. For generating the lexical analyzer, we chose Flex, because it generates faster scanners than Lex, and it has the start condition feature (see Section 3.1) that Lex lacks.

[3] LALR(1) is the class of languages that can be parsed from Left to Right with a Look-Ahead of 1 symbol.[1]

[4] If the grammar is really complex, such as the grammar for ASN.1 [26] with the parts involving user-definable syntax, hand-coding the parser may be the only feasible choice. Luckily SDL is much easier for the compiler writer in this aspect.

1.2.2 Programming Language: C++ and STL

The choice of programming language was easy. The only truly portable[5] and widely available, efficient programming languages are C [4] and C++ [5]. The programming language C, originally developed in the 1970s, sometimes referred to as "high-level Assembler,"[6] is available for practically every computer system, and C++, being the "official successor" of C, is fairly widely available,[7] sometimes even as a standard component of the operating system. Of the two languages, C++ was chosen, because it encourages the programmer to write modular and maintainable code and contains constructs needed in object-oriented programming.

Compilers involve complex data structures. We decided to use the Standard Template Library (STL), which has C++ class templates for a wide range of basic building blocks of data structures, ranging from simple lists to associative arrays and hash tables.

The Standard Template Library depends on many C++ constructs that have been added to the language very recently. For this reason, only the newest C++ compilers can compile the SDL front-end. We do not see this as a problem, since the freely available EGCS C++ compiler [14], which generates code for practically every modern computer platform, supports the STL well enough for our purposes. The current version of C++ [5] recently changed its status from a draft to an international standard, and all major C++ compilers are likely to follow it by the time the whole analyzer is ready to be released.

1.2.3 Version Control System: CVS

Often there are cases where one makes a modification to a file and regrets it later, wanting to revert to an earlier version of the file. The problem could be solved by backing up each file before it is modified, but this solution is inefficient if the modifications are small compared to the size of the file. A *version control system* is a much better choice.

A version control system records the modification history of files. When a file is kept in a version control system, it is possible to retrieve any previously stored version of the file and to see the differences between any versions.

There are two freely available version control systems: RCS [22] and CVS [2]. We chose CVS, because it has been designed for distributed setups and does not require files to be locked for modification. The database, called *repository*, is kept in one place, and files can be retrieved from it even across the network. Even when the development takes place on several workstations, only the repository needs to be backed up. In the case of a disk crash, only those modifications that had not yet been checked in to the repository will be lost.

[5] A portable program developed for one computer system using one standard-conforming compiler can be compiled for (ported to) any system using some other compiler.

[6] This is a two-edged sword: C is an extremely efficient language with as few restrictions as possible, but writing code in C is error-prone.

[7] The C and C++ compilers from the GNU project are freely available for practically every modern all-purpose computer.

1.3 Outline of the Thesis

This work is divided into chapters as follows. Chapter 2 discusses the problems involved with language recognition, called *parsing*, and shows how these problems can be solved when parsing SDL. Chapter 3 presents the problem field related with *lexical analysis*, separating characters of a source language into groups that logically belong together. Chapter 4 shows how syntax errors are handled by the compiler, and Chapter 5 presents the data structures our compiler uses for storing SDL programs and discusses some of the semantic analysis undertaken by the compiler. Finally, Chapter 6 discusses the debugging and testing methods that have been used while developing the compiler, and Chapter 7 concludes the work and looks at areas of future work.

Chapter 2

Implementing the Grammar

Implementing a large grammar such as the one for SDL is all but straightforward. The standard [27] extensively tries to express within the grammar even such constraints that are intuitively semantical. This leads to ambiguities, meaning that some constructs of the language can be parsed in several alternative ways using different grammar rules. Even after removing these ambiguities, for example after omitting the grammatical distinction between constant expressions (called *ground expressions* in SDL) and more general expressions, there are plenty of situations that can only be solved with extensive modifications to the grammar or with techniques that extend the look-ahead in the lexical analyzer, which are discussed in Sections 2.2.2 and 3.2.

2.1 Introduction to LALR(1) Parsers

Programming languages are usually defined in terms of a *context-free grammar*, which gives a precise, yet easy to understand, syntactic specification for the language. A properly designed context-free grammar makes it possible to automatically construct an efficient parser for the language.

Bison and Yacc are LALR(1) parser generators, meaning that they can generate parsers for languages that can be parsed by reading them from left to right, with all decisions being based on one look-ahead symbol. Different classes of languages are presented in [1].

LALR(1) parsers work by reading the input, consisting of *terminal symbols*, to a stack, symbol by symbol. The terminal symbols, also called *tokens*, are delivered to the parser by a lexical analyzer (see Chapter 3), which converts the input, a stream of characters, to a stream of tokens, or terminal symbols. Symbols may have attributes. For instance, a token representing a variable name in a programming language could have a character string attribute that represents the variable name.

Reading terminal symbols to the parser is called *shifting*. As soon as one or more symbols on the top of the stack match the right-hand-side of a grammar rule, they will be replaced with the left-hand-side of the rule, which is a single

non-terminal symbol.[1] This is called *reducing*. When the whole input has been read, there should be only one (non-terminal) symbol on the stack, called the *start symbol*. Parsers working by shifting and reducing are often called *shift-reduce* parsers.

2.1.1 Notation

SDL is defined in [27] with a context-free grammar, also called a Backus–Naur Form (BNF) description. This thesis uses a notation similar to [27]. All terminal symbols are presented like `this` and all non-terminal symbols like ⟨*this*⟩, while ITU-T uses **bold** typeface for keywords and ⟨symbol⟩ or ⟨attribute symbol⟩ for other symbols. In our notation, we omit symbol attributes. Also, whenever we present SDL grammar rules, we often omit uninteresting alternatives and intermediate rules from them for clarity. For the sake of readability, we use the notation normally used for non-terminal symbols for some terminal symbols that are not keywords, mainly for the ⟨*name*⟩, ⟨*quoted operator*⟩ and ⟨*character string*⟩ tokens.

A context-free grammar consists of a left-hand-side symbol and right-hand-side that can be a sequence of terminal or non-terminal symbols or the empty sequence (here marked with ε). For instance, to describe a language that consists of any number of terminal symbols `id` separated by commas, one could write the following grammar:

$$\begin{aligned}
\langle S\rangle &:= \varepsilon \\
\langle S\rangle &:= \langle s\rangle \\
\langle s\rangle &:= \texttt{id} \\
\langle s\rangle &:= \langle s\rangle\texttt{,id}
\end{aligned}$$

In order to compact the notation, we introduce the | operator, which is used for representing alternatives, and apply it to the grammar:

$$\begin{aligned}
\langle S\rangle &:= \varepsilon \\
&\mid \langle s\rangle \\
\langle s\rangle &:= \texttt{id} \\
&\mid \langle s\rangle\texttt{,id}
\end{aligned}$$

This still looks rather complicated. We introduce the Kleene closure operator: $\langle A\rangle := \langle a\rangle^*$ means that $\langle A\rangle$ expands to any number of occurrences of $\langle a\rangle$ (including zero occurrences):

$$\begin{aligned}
\langle A\rangle &:= \varepsilon \\
&\mid \langle A\rangle\,\langle a\rangle
\end{aligned}$$

[1] Non-terminal symbols can replace or be replaced with sequences of symbols unlike terminal symbols, which are delivered by the lexical analyzer.

With $\langle B \rangle := [\langle b \rangle]$ we denote that $\langle B \rangle$ expands either to the empty sequence or to $\langle b \rangle$:

$$\begin{aligned} \langle B \rangle &:= \varepsilon \\ &\mid \langle b \rangle \end{aligned}$$

Now the grammar for $\langle S \rangle$ can be written in a compact form, similar to the Extended Backus–Naur Form (EBNF) suggested in [25]:

$$\langle S \rangle := [\texttt{id}(\texttt{,}\ \texttt{id})^*]$$

Here we use round parentheses in normal typeface $()$ for grouping grammar symbols.

2.1.2 Attributes and Semantic Actions

In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In Bison, a grammar rule can have an "action" made up of C or C++ statements. Each time the parser recognizes a match for that rule, the action is executed.

As mentioned in the beginning of this chapter, grammar symbols may have attributes attached to them. For instance, a terminal symbol representing an integer literal could have the numeric value of the integer as an attribute.

Consider the simple example presented in Figure 2.1. It demonstrates how semantic actions and symbol attributes, also called semantic values, can be used to calculate the value of expressions during parsing. The example makes use of *synthetic attributes*: the semantic value of the `expr` on the right-hand-side is not known until the parser has reduced something to `expr`. Actions in Bison typically compute (synthesize) the semantic value of the left-hand-side symbol of a rule from the semantic values of the right-hand-side symbols.

If an attribute of a non-terminal symbol on the right-hand-side of a grammar rule is computed using an attribute of the left-hand-side symbol, it is called an *inherited attribute*. Because using this type of attributes is very difficult in Bison, Yacc and many other parser construction tools, they are rarely used.

2.1.3 Conflicts

Not every language for which a context-free grammar exists can be handled by a shift-reduce parser. There can be situations when the parser does not know whether

```
expr : expr '+' term { $$ = $1 + $3; }
     | term { $$ = $1; };
```

Figure 2.1: Using attributes and semantic actions in Bison

it should shift or reduce, or which reduction it should perform. These situations are called *conflicts*, and they are a property of the grammar.

Conflicts in a grammar cause trouble in parsing, i.e. when we want to determine whether a particular input string belongs to the language defined by the grammar. Conflicts that cannot be easily removed by altering the grammar must be studied carefully, to determine what influence they have on the language recognized by the parser or on the semantic actions of the parser.

Reduce-Reduce Conflicts

A *reduce-reduce conflict* occurs when the symbols on the stack match several rules at the same time, i.e. when two or more rules apply to the same sequence of input. Usually this kind of conflicts can be detected directly by looking at the grammar rules, and in practice, they are easier to remove than shift-reduce conflicts.

The grammar in Figure 2.2 exhibits a reduce-reduce conflict. The input consisting of a single `term` can be parsed in two ways. It could be reduced to ⟨*item*⟩ and then to ⟨*sequence*⟩ via the second rule. Alternatively, the empty sequence could be reduced into a ⟨*sequence*⟩ via the first rule, and this could be combined with the `term` using the third rule for ⟨*sequence*⟩. In a similar manner, an empty input can be reduced to ⟨*sequence*⟩ in two ways. This example is from the Bison documentation [6].

When there is a reduce-reduce conflict, the parser generator will systematically choose one of the possible reductions for all the input, which could be e.g. the reduction whose definition comes first in the grammar. In our simple example this seems harmless, but when the rules are associated with different semantic actions, the difference becomes significant.

Figure 2.3 shows how the ambiguities can be removed from our example. The semantic actions of the rules must still handle all the cases involved with the orig-

$$
\begin{aligned}
\langle \textit{sequence} \rangle &:= \varepsilon \\
&\mid \langle \textit{item} \rangle \\
&\mid \langle \textit{sequence} \rangle \texttt{term} \\
\langle \textit{item} \rangle &:= \varepsilon \\
&\mid \texttt{term}
\end{aligned}
$$

Figure 2.2: An ambiguous grammar for a sequence of symbols

$$
\begin{aligned}
\langle \textit{sequence} \rangle &:= \varepsilon \\
&\mid \langle \textit{sequence} \rangle \texttt{term}
\end{aligned}
$$

Figure 2.3: Disambiguated version of the grammar in Figure 2.2

inal grammar. This is easiest to accomplish by using the same data structure both for ⟨*item*⟩s and for sequences of ⟨*term*⟩.

Shift-Reduce Conflicts

Another type of conflicts are *shift-reduce conflicts*, which occur when both shifting and reducing is possible, i.e. when a set of symbols on the top of the stack matches the right-hand-side of a rule while another rule's right-hand-side partially matches (a possibly different) set of symbols on the top of the stack.

Consider the simple grammar presented in Figure 2.4 and an input consisting of the tokens `term + term * term`. After the parser has shifted in the first symbol `term`, reduced `term` → ⟨*expr*⟩ and shifted `+ term` (again reducing `term` → ⟨*expr*⟩), it has a choice between reducing

$$\langle expr \rangle + \langle expr \rangle \rightarrow \langle expr \rangle$$

and shifting the `*`, since the rule

$$\langle expr \rangle := \langle expr \rangle \texttt{ * } \langle expr \rangle$$

partially matches the ⟨*expr*⟩ on the top of the stack combined with the look-ahead symbol `*`. This is a shift-reduce conflict. With the additional information that the `*` operator has precedence over the `+` operator, the parser decides to shift. It will shift the `*` and the third `term`. At this point, the stack will be reduced as follows:

$$\begin{aligned} \langle expr \rangle + \langle expr \rangle \texttt{ * term} \\ \langle expr \rangle + \langle expr \rangle \texttt{ * } \langle expr \rangle \\ \langle expr \rangle + \langle expr \rangle \\ \langle expr \rangle \end{aligned}$$

The rightmost symbols are on the top of the stack. The states of the parser are not shown in this example.

Bison and Yacc can resolve this kind of conflicts with the help of operator precedences [1, pp. 203–215]. However, often shift-reduce conflicts are caused by rules that seem totally unrelated with each other, and they can only be resolved in

$$\begin{aligned} \langle expr \rangle &:= \langle expr \rangle \texttt{ + } \langle expr \rangle \\ \langle expr \rangle &:= \langle expr \rangle \texttt{ - } \langle expr \rangle \\ \langle expr \rangle &:= \langle expr \rangle \texttt{ * } \langle expr \rangle \\ \langle expr \rangle &:= \langle expr \rangle \texttt{ / } \langle expr \rangle \\ \langle expr \rangle &:= \texttt{(} \langle expr \rangle \texttt{)} \\ \langle expr \rangle &:= \texttt{term} \end{aligned}$$

Figure 2.4: A simple grammar for expressions

⟨*entity in package*⟩ := ⟨*package reference clause*⟩*
SYSTEM TYPE ⟨*identifier*⟩
[⟨*formal context parameters*⟩]
[⟨*specialization*⟩] ⟨*end*⟩
⟨*entity in system*⟩*
ENDSYSTEM TYPE [⟨*identifier*⟩] ⟨*end*⟩
| SYSTEM TYPE ⟨*name*⟩ REFERENCED ⟨*end*⟩

Figure 2.5: SDL grammar rules related with system types

the hard way by rewriting parts of the grammar in a way that makes parsing easier but does not alter the language defined by the grammar.

Any two grammars define the same language if the start symbol in both grammars expands to the same set of sequences of non-terminal symbols. When part of the grammar is modified, one can ensure that the language defined by the grammar is not affected by ensuring that all non-terminal symbols whose rules refer to a symbol affected by the modification continue to expand in the same way as earlier. Performing the comparisons manually is somewhat error-prone, and it is a good idea to run extensive tests on the parser once it has been finished, involving as complex input as possible.

Bison and Yacc resolve all remaining shift-reduce conflicts by deciding to shift.

2.2 Implementing the SDL Grammar

The SDL grammar has many ambiguities that must be removed before any parser can be generated from it. The most common ambiguity involves names and identifiers. In SDL, identifiers are names preceded by an optional qualifier. Most rules in the SDL grammar accepting identifiers have alternatives that accept names. These rules cause reduce-reduce conflicts, which can be removed by removing the alternatives allowing only names, causing the names to be parsed as identifiers.

It was quite easy to remove reduce-reduce conflicts from the SDL grammar, since not many of them were more complicated than the example presented in Figure 2.2. Shift-reduce conflicts caused much more trouble, and some of them could only be removed by modifying the lexical analyzer, as discussed in Section 2.2.2.

2.2.1 Relaxing the Grammar

Many shift-reduce conflicts are caused by rules that look similar but begin differently. For instance, consider the rule set presented in Figure 2.5.

There are two conflicts. One can be removed by replacing the ⟨*name*⟩ in the second rule with ⟨*identifier*⟩. The other conflict is caused by the non-terminal symbol ⟨*package reference clause*⟩* being present in the first rule and absent in the sec-

ond rule. After adding the non-terminal to the second rule, both rules begin in the same way, and since neither [⟨*formal context parameters*⟩] nor [⟨*specialization*⟩] contains the terminal symbol `REFERENCED`, the rule set will be free of ambiguities. Of course now the semantic actions of the second rule must ensure that the ⟨*package reference clause*⟩* was reduced from an empty sequence and that the ⟨*identifier*⟩ actually is a ⟨*name*⟩.

Modifying the grammar so that it accepts a larger language than specified in the standard actually serves the user. Instead of reporting a "parse error," the parser can report that package reference clauses make no sense with system type references, or that system types can only be referenced by a name, not by an identifier. In this case we were forced to relax the grammar, but the grammar can also be relaxed on purpose; see Section 4.2.

For the record, our lexical analyzer recognizes the SDL keyword pair `system type` as a single token `SYSTEMTYPE`, because converting the keywords to two tokens would introduce conflicts elsewhere in the grammar.

2.2.2 Extending the Look-Ahead

Not all shift-reduce conflicts can be disposed of by relaxing the grammar in a feasible way. LALR(1) parsers make all decisions using at most one look-ahead symbol, which is a restriction. Since Bison does not support longer look-aheads, the only place where the look-ahead can be extended is the lexical analyzer.

Rule sets requiring a longer lookahead than one symbol are usually handled by replacing a terminal symbol in one or more rules with a new token and by modifying the scanner accordingly so that it will return the new token when the symbol is followed by anything that only matches the rules containing the new token.

In our SDL parser, all look-ahead extensions are related with keywords. For instance, the `operators` keyword can be converted to an `OPERATORS` or to an `OPERATORS_` token, depending on the input following it. Section 3.2 describes the extended look-aheads implemented in the lexical analyzer.

2.2.3 Modifications to the Language

Some constructs of the SDL grammar are difficult to implement in an LALR(1) parser. The SDL parser developed at Humboldt-Universität zu Berlin [16] does not implement the most difficult parts of the grammar. It omits some parts, like ⟨*select definition*⟩, that are problematic at the semantic level, and modifies some language constructs that are extremely difficult to parse.

We put considerable effort into avoiding any modifications to the language recognized by our SDL parser. We only gave up on one thing: macros. They could be expanded by a separate preprocessor à la C at some later point.

For the rules presented in Figure 2.6 we could not come up with any reasonably simple look-ahead extension schemes. The symbol ⟨*formal context parameter*⟩, which is used in an ⟨*end*⟩-separated list, has one right-hand-side ending in the non-terminal symbol ⟨*process signature*⟩ and another ending in ⟨*procedure signature*⟩.

$$
\begin{array}{rcl}
\langle \textit{process signature} \rangle & := & \varepsilon \\
& | & [\langle \textit{end} \rangle] \; \textsf{FPAR} \; \langle \textit{sort} \rangle \; (\texttt{,} \; \langle \textit{sort} \rangle)^* \\
\langle \textit{procedure signature} \rangle & := & \varepsilon \\
& | & [\langle \textit{end} \rangle] \; \textsf{FPAR} \; \langle \textit{parameter kind} \rangle \langle \textit{sort} \rangle \\
& & (\texttt{,} \; \langle \textit{parameter kind} \rangle \langle \textit{sort} \rangle)^* \\
& & [\langle \textit{end} \rangle \; \textsf{RETURNS} \; \langle \textit{sort} \rangle] \\
& | & [\langle \textit{end} \rangle] \; \textsf{RETURNS} \; \langle \textit{sort} \rangle
\end{array}
$$

Figure 2.6: SDL grammar rules containing troublesome ⟨*end*⟩ symbols

$$
\begin{array}{rcl}
\langle \textit{variables of sort} \rangle & := & \langle \textit{name} \rangle \; [\textsf{AS} \; \langle \textit{identifier} \rangle] \; (\texttt{,} \; \langle \textit{name} \rangle \; [\textsf{AS} \; \langle \textit{identifier} \rangle])^* \\
& & \langle \textit{sort} \rangle \; [\texttt{:=} \; \langle \textit{expression} \rangle]
\end{array}
$$

Figure 2.7: Variable definitions in SDL

The parser can by no means determine whether an ⟨*end*⟩ symbol is a list delimiter or part of ⟨*formal context parameter*⟩. We came up with a similar solution as the one used in [16]: we removed the ⟨*end*⟩ symbols from the rules in Figure 2.6. Since this restricts the grammar, we added the ⟨*end*⟩-less ⟨*process signature*⟩ and ⟨*procedure signature*⟩ rules directly to the rule ⟨*formal context parameter*⟩.

For now, the semantic actions of these rules only issue error messages, but they could do better: they could check whether a previously parsed construct ⟨*formal context parameter*⟩ can be extended with the *FPAR* or *RETURNS* construct, and issue the error message only when the construct is not allowed. After this, the semantic action of the second rule of ⟨*procedure signature*⟩, whose optional `returns` part now makes the parser slightly incompatible with [27], could be modified to issue a warning message if the `returns` keyword is present, saying that a semicolon is expected before the keyword. Since these grammar rules only apply for seldomly used constructs, we decided to skip the fine-tuning at this point.

2.2.4 Semantic Actions

Bison does not support inherited attributes very well. They can only be used by accessing the symbol stack with negative indices, which is very hazardous and may be impossible if the same rule occurs on the right-hand-side of several different rules. We chose to limit ourselves to using synthesized attributes, and implemented everything else with the help of global variables. Sometimes global variables were used even when using synthesized attributes would have been possible. This reduces the amount of attribute types needed (the `%union` definition in Bison) and the need of post-processing the parsed input.

Sometimes post-processing the parsed input cannot be avoided. For instance,

the grammar rule covering SDL variable definitions in Figure 2.7 first expects the variable names and then their sort. For the parser it would be more convenient to know the sort in advance, in which case each defined variable could be assigned a sort already at the point when the variable name is read. With an SDL-like grammar, the names of the variables must be kept in a temporary storage until their type (called *sort* in SDL) is known. Actually, since SDL does not forbid forward references, the semantic value of the non-terminal ⟨*sort*⟩ is not a reference to a type but an identifier that cannot be resolved until the whole SDL specification has been parsed.

Global Variables in the SDL Parser

In addition to the variables updated by the lexical analyzer, `lineno` and `filename`, which identify the definitions being currently parsed, the parser makes use of the following global variables.

`bool quoted`
: The grammar rule ⟨*name*⟩ covers names and quoted operator names. This flag is used to determine whether the last ⟨*name*⟩ read was a name or a quoted operator name. This variable makes it possible to tolerate quoted operators in the place of names or vice versa, without introducing any ambiguities to rules accepting both names and quoted operators.

`bool generator_actual`
: The rule ⟨*generator actual*⟩ contains overlapping definitions. One right-hand-side symbol of the rule, ⟨*term*⟩, partially overlaps with other right-hand-sides of ⟨*generator actual*⟩. This is set to `true` whenever the grammar rule ⟨*generator actual*⟩ is active; it will be set to `false` by the actions in ⟨*term*⟩ as soon as it becomes clear that the ⟨*term*⟩ cannot be any other ⟨*generator actual*⟩.

`bool reverseSignal`
: In ⟨*signal refinement*⟩, the non-terminal symbol ⟨*signal definition*⟩ can be preceded by the `reverse` keyword. If the keyword is present, this flag will be set. The flag is used in the semantic action of ⟨*signal definition*⟩, and it can be seen as an inherited attribute in the grammar.

`enum RevealedExported revealed`
: The `revealed` and `exported` keywords can precede the variable names in a ⟨*variable definition*⟩. This global variable keeps track which keywords are present, and it is used by the rule ⟨*variables of sort*⟩. The variable was introduced for purely cosmetic reasons, to reduce the size of the `%union` definition in the parser specification.

`class State* currentState`
: The SDL grammar rules related to state automata span a parse tree that is quite different from any straightforward data structure for describing state

automata. This variable holds a pointer to the state definition that is being parsed.

`stack<Entity*,list<Entity*> > ent`
: This variable holds the SDL entity stack. In SDL, entities such as block, process and channel, are nested inside each other, and each entity has its own scope of declarations. The currently active entity is kept on the top of this stack.

In addition to these variables, the Bison input file defines many auxiliary functions. The following functions allocate objects; others mainly take care of type conversion, error reporting and some semantic checking.

`FparList* currentParams()`
: When first called, this function returns a new formal parameter list. On subsequent calls, it returns a reference to the same list. This function is called by the semantic actions of ⟨*formal context parameters*⟩ list items. Whenever a list item, defined by ⟨*formal context parameter*⟩, is parsed, its semantic value will be appended to the same list, maintained by the function `currentParams()`. In this way, we avoid having to associate a semantic value to each kind of ⟨*formal context parameter*⟩ in the Bison specification; the semantic values will be internal to the semantic actions. This reduces the size of the `%union` definition.

`FparList* ResetParams()`
: This function returns a reference to the list most recently allocated by the previously described `currentParams()` function (`NULL` if no list was allocated) and resets the list, so that the next call to `currentParams()` will allocate a new list. The function will be called after ⟨*formal context parameters*⟩ has been parsed.

`SignalList* currentSignals()`
: This function is otherwise the same as `currentParams()`, but it operates on signal lists.

`SignalList* ResetSignals()`
: This function pairs with `currentSignals()` in a similar manner as the `ResetParams()` function pairs with `currentParams()`.

Semantic Checks instead of Syntactic

Programming language grammars found in standards or user manuals often try to incorporate as many intuitively semantical constraints as possible. For instance, the grammar might distinguish between constant expressions (⟨ground expression⟩ in [27]) and generic expressions (⟨expression⟩). Some people may find such grammars easier to read, others think that a few sentences of text is more understandable, like this: “Numeric constants always are constant expressions. Function

calls never are constant expressions. Other expressions that only consist of constant expressions are constant expressions." For a compiler writer, the latter way is preferable, because overlapping definitions lead to ambiguities very easily. Therefore, we find it desirable that such matters as differentiating between constant and non-constant expressions are handled by separately implemented semantic checks rather than by syntactic ones built in the grammar.

Semantic checking is also an advantage for the compiler user, because a compiler applying it can generate more helpful error messages. Instead of telling the user that his or her program violates the grammar, the compiler can say, for instance, that SDL variables can only be initialized with constant expressions. Chapter 4 discusses how the error reports of a compiler can be made more user-friendly.

Chapter 3

Lexical Analysis

Usually there is a clear distinction between lexical analysis and parsing. The lexical analyzer, also called *scanner*, converts a stream of characters to a stream of named tokens. For instance, the input `13+456` could be converted to a token stream *`NUMBER PLUS NUMBER`*, with integer attributes attached to the *`NUMBER`* tokens. These tokens usually have a direct correspondence with the terminal symbols in the grammar. When shifting, the parser fetches one token from the scanner. The scanner does not need to convert each input string to a token. Space, tabulator and newline characters (often called *whitespace*) and comment strings are usually silently discarded by the scanner.

GNU Flex is the best established scanner generator that generates C or C++. The relation between character strings and tokens is specified with search patterns expressed with *regular expressions* and actions attached to each search pattern. An introduction to regular expressions and scanners can be found in [1].

Flex translates each regular expression into a finite automaton that determines whether the regular expression matches the string of characters read so far. The automata use *greedy* matching, meaning that they search for the longest possible input strings that match the regular expressions. All automata corresponding to the regular expressions are executed in parallel, and as soon as a maximal match for the input string is found, the action attached to the matched regular expression will be executed. This action is a C or C++ block, and it usually ends in a `return` statement that returns a token to the parser. If several regular expressions match the input, Flex chooses the regular expression that comes first in the specification.

The SDL standard [27] does not define lexical rules in terms of regular expressions, but with grammar rules. Converting them to regular expressions is all but trivial. The biggest problems are related to the underscore character. In SDL, names consist of one or more words separated with single underscores, and words consist of alphanumeric characters and dots, but each word has to contain at least one alphanumeric character. The following quote from [27] defines yet another constraint:

> When an ⟨underline⟩ character is followed by one or more ⟨space⟩s, all of these characters (including the ⟨underline⟩) are ignored, e.g. `A_ B` denotes the same ⟨name⟩ as `AB`. This use of ⟨underline⟩ allows ⟨lexical unit⟩s to be split over more than one line.

Since ⟨lexical unit⟩ means any token recognized by the lexical analyzer, also keywords and other special tokens that appear as ⟨name⟩ can be split over multiple lines. All such tokens must be first detected as name, and before returning any *NAME* to the parser, the scanner has to check whether it is a reserved word in SDL, and return the corresponding token if it is. This means that dealing with reserved words, in particular, is much more complicated than in most other programming languages.

The problematic underscore characters followed by whitespace could also have been processed by using a separate preprocessor or by chaining two scanners, the first of which would filter out the underscores and the whitespace following them. Had we implemented an SDL macro preprocessor, we could also have handled the underscores there, which would have simplified the actual scanner and made it somewhat faster.

The most difficult problems encountered while developing the lexical analyzer for SDL involved the handling of names and underscores and the various look-ahead extensions required to make the Bison-generated parser work with a look-ahead of only 1 token.

3.1 Start Conditions

Flex translates search patterns into finite automata. In the domain of automata it is natural to speak about states. Introducing the concept of states to the search patterns seems possible without any performance penalty. This has been done in Flex, where the search pattern "states" are called *start conditions* of regular expressions.

Start conditions divide the search patterns of the scanner specification into subclasses. Normally, all search patterns are active simultaneously. Start conditions limit the amount of active search patterns. The scanner will use only the search patterns associated with the currently active start condition. This is how *exclusive* start conditions work in Flex. When an *inclusive* start condition is active, also rules within the initial start condition are enabled. From now on we assume that all start conditions are exclusive. Two typical tasks for start conditions are parsing string constants and stripping comments.

3.1.1 An Example: Handling Comments

Start conditions not only make regular expressions more readable; they also make the generated scanners more efficient. Consider the lexical comments in SDL. They begin with "`/*`" and end in "`*/`", and they can span over multiple lines. So, they can be matched with the regular expression `/\*([^*]|\*[^/])*\*/`.[1] Scanners generally keep track of line numbers, because they are extremely useful in diagnostic messages. The scanner will have to count the newline characters also

[1] The character class complementation (e.g. `[^/]` for matching any character but "`/`") is used, because otherwise the greedy pattern matching applied by Flex could skip the end of a comment and ignore anything up to the end of the last comment in the file.

```
%x comment
%%
int line_num = 1;

"/*"                    BEGIN(comment);

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

Figure 3.1: Flex definitions for stripping C-style comments

inside the comment string, since it is not acceptable for the line counter to become out of synchronization in the presence of multi-line comments.

The approach of processing comments with one regular expression is inefficient. Not only does it require the comment to be processed twice, but the comment string must also be copied and preserved in the memory until the whole comment has been read. Start conditions allow the scanner to count lines while skipping comments. For multi-line comments, the matched strings will be shorter, since only one line will be read at a time. This reduces the memory requirements. The definitions in Figure 3.1 are from the Flex documentation [18].

3.1.2 Start Condition Stack

When the scanner specification contains lots of start conditions and when there are "sub-automaton calls" in the Flex specification, the `%option stack` feature of Flex becomes very useful. When this option is enabled, some functions for handling a start condition stack become available in Flex actions. We make use of the function `yy_push_state(s)`, which pushes the current start condition on the stack and switches to a new start condition `s`, and of `yy_pop_state()`, which returns from the "sub-automaton call." The `BEGIN(s)` macro can be used for switching start conditions without affecting the stack, i.e. as a "goto" statement.

Scanner look-ahead techniques, which will be discussed in Section 3.2 in detail, often require start conditions whose only action returns an already scanned token to the parser, no matter what the next input character is. Such behavior can be achieved with the pattern `.|\n`, which matches any character, including the newline character.[2] Since this action cannot do anything with the character the dummy pattern matches, it puts it back to the input buffer by calling the `yyless()` function.

[2] End-of-file conditions still need to be handled separately.

3.1.3 Basic Start Conditions in the SDL Scanner

The SDL scanner has relatively many start conditions. Most of them are related to look-ahead techniques (see Section 3.2). In this section we will only describe the basic start conditions.

Comments and Include Files

SDL supports two kinds of comments. The graphical representation of SDL has a special comment symbol, whose counterpart in SDL/PR is the `comment` keyword followed by a character string. This kind of comments is handled by the parser. The other kind are C-style lexical comments, called ⟨note⟩ in [27], which are allowed anywhere between lexical symbols.

In some languages, comments are often misused for passing extra information to the compiler, because the language lacks constructs for otherwise passing that information. As a non-standard addition, many SDL-related tools support *include files* in a slightly different way than the C preprocessor does. The comment `/* #include 'file.pr' */` will cause the scanner to open a file named `file.pr`, to use the file as input, and to switch back to the original input file upon reaching the end of the include file.

In order to be compatible with other SDL tools, our scanner contains support for include files. This is achieved by extending the search patterns for skipping comments and by adding two start conditions:

comment
: This start condition will be called when the beginning of a comment is seen in the input.[3] Control will return to the calling start condition when the scanner finds the end of a comment. Comments are not blindly skipped; the scanner keeps an eye on letters preceded by a number sign (#), and whenever it encounters the string `#include` or `#INCLUDE` in the comment, it makes a sub-automaton call to include1.

include1
: This start condition will be called when an `#INCLUDE` directive is seen in a comment. Whitespace will be skipped, and if an apostrophe is seen, the scanner will switch to include2 and call sdlstring, which will read the file name. Any other character will cause a return back to comment.

include2
: When the include file name has been read by sdlstring, this start condition will be activated. It will switch buffers and switch to the initial start condition, effectively inputting the specified file at that place. Upon reaching the end of the include file, the scanner will return to the calling start condition (comment).

[3] The search pattern for start of comment is disabled when the scanner is processing comments, names or strings, or when the active start condition does not skip whitespace.

Strings

As a compensation for having C-style comments, SDL has Pascal-like character string constants. Strings are delimited by apostrophes, and apostrophes occurring in string constants are represented with two successive apostrophes. That is, `'ab''c'''` represents the string constant `ab'c'`.

The ITU-T Recommendation Z.100 [27] effectively limits the character set allowed in SDL string constants to the International Alphabet 5 [21] codes 32–126, but we chose to remove this arbitrary-feeling limitation. Our scanner allows all 8-bit character codes 0–255 in string constants, and it is up to the user to obey the limitations. Even the NUL character (code 0) can be allowed, since character strings in SDL carry length information with them and need not be NUL-terminated like e.g. strings passed to C run-time library functions.

The start condition sdlstring is called from the initial start condition and some other places when an apostrophe is seen in the input. In sdlstring, the apostrophe and newline characters are treated differently from other characters, which are simply appended to the buffer holding the string constant scanned so far. An occurrence of two successive apostrophes causes the scanned string to be extended by one apostrophe. When the scanner sees a newline character, it will update the line counter. Control will return to the calling start condition when a single apostrophe occurs in the input.

Quoted Operators

In SDL, operators can be defined for user-defined data types. When defining any built-in operators of SDL, the operator names must be surrounded by double quotation marks in order to avoid confusion. Because double quotation marks are not used anywhere else in the SDL grammar, quoted operators can be handled by the scanner without fear of breaking anything.

Handling ⟨*quoted operator*⟩ in the scanner is efficient, because the parser is given only one token instead of three, and there is no need to convert the operator names to tokens and back to strings. Also errors can be tolerated better in this approach. The scanner can return a special token *ERR_OP_NAME* instead of the normal *OP_NAME* if something else than a valid operator name is quoted, or if the closing quotation mark is missing.

Scanning quoted operator names can be implemented with two start conditions, both of which skip whitespace and comments in the normal way.

op1

> This start condition will be called when a double quotation mark (") is detected in the input. If anything that looks like an SDL name is encountered in this start condition, the scanner will switch to op2 and call word1 or word2. When seeing SDL operator names consisting of special characters, the scanner switches to op2. The scanner will return from op1 and return an *ERR_OP_NAME* token to the parser if it encounters a closing quote or any unexpected character (which will be put back to the input buffer).

op2

When the scanner reaches this start condition, it will have read a double quotation mark followed by an operator name or other name. After skipping potential comments and whitespace, the scanner is expecting a double quotation mark. If one is found in the input, the name scanned will be checked. If it is a malformed name or not a built-in operator name, again *ERR_OP_NAME* will be returned to the parser. The same happens if any other character than a double quotation mark is read (and put back to the input buffer). In any case, the scanner will return to the start condition that originally called op1.

Names and Reserved Words

SDL has very complicated rules for names. Because also literals of numeric types are names, names may contain dots and even begin with them. Underscores have a special position: names consist of one or more words combined with underscore characters. More than one successive underscore character in a name is an error, and the words must contain at least one alphanumeric character. This is not all: underscore characters followed by whitespace or comments will be ignored, i.e. "A_ B" is the same as "AB". The current SDL standard even allows underscores in names to be replaced with whitespace, but this has not been implemented in our compiler, because doing so would introduce lots of ambiguities to the grammar or require infeasibly complex look-ahead techniques.

SDL names could be matched with one rather complex regular expression, but we decided to use a start condition based approach for three reasons. First, using them makes it possible to detect (and tolerate) malformed SDL names. Second, this approach avoids backtracking. The `flex -b` option reports whether the scanner automaton ever has to backtrack, after making a wrong guess about a search pattern that first seems to match the input. Dividing the rather complex regular expression into several subexpressions using start conditions makes it possible to completely avoid backtracking, which can make the scanner more efficient. Third, there is no need to postprocess scanned names, because the line number counter can be updated while scanning, and parts consisting of underscore characters followed by whitespace or comments can be skipped.

Four start conditions are needed. Two of them are related with ⟨word⟩s, which must contain at least one alphanumeric character, and the other two are related with the underscore character. To keep the following description simple, we do not say anything about lexical comments, which can occur before and after underscores in SDL names.

word1

This start condition is called when a "." occurs in the input. Any further dots read from the input will be appended to the name string. An alphanumeric input will cause a switch to the word2 state, and an underscore followed by whitespace will cause a call to underscore_ws. An occurrence of a sole underscore character will cause the name to be marked malformed (since it

contains a component with no alphanumeric characters) and the start condition to be switched to underscore. If the scanner sees any other character, it will put back the character, return to the calling start condition and mark the name malformed (because the last word read contains no alphanumeric characters).

word2
: When this start condition is called or switched to, the currently read word is known to contain alphanumeric characters. The actions are otherwise identical to word1, but there is no transition to word2, and the name will never be flagged malformed.

underscore
: This start condition is switched to after reading an underscore not followed by whitespace. A well-formed SDL name must continue with dots (which causes a switch to word1) or with alphanumeric characters (switch to word2). Underscores in this start condition cause the name to be flagged malformed, since it contains several underscore characters in a row. Underscores followed by whitespace will cause a call to underscore_ws. Any character unmatched by the other rules will be put back to the input, and after seeing such a character the scanner will return to the calling start condition and flag the name malformed, because its last word component is empty.

underscore_ws
: This start condition is called when an underscore character followed by whitespace is seen. All succeeding whitespace will be skipped, as will underscore followed by whitespace. A sole underscore character not followed by whitespace or a sequence of dots causes a return to the calling start condition, and alphanumeric characters in the input make the scanner remove the calling start condition from the stack and switch to word2, since the next word component is guaranteed to be well-formed.

The names scanned will usually be returned to the parser through the start condition ret, which checks whether the name is an SDL keyword and calls keyword-related start conditions (see Section 3.2) if necessary. If the name is malformed or not a keyword, the string representing it will be copied from the string buffer of the scanner, and a *NAME* or *ERR_NAME* token will be returned to the parser. Otherwise the token corresponding to the SDL keyword will be returned. This approach makes it possible for the scanner to support keywords written in mixed case. The next version of SDL should support only keywords written entirely in upper case or lower case, and this is also what our scanner normally does.

3.2 Extending the Look-Ahead

Some SDL grammar rules are ambiguous for LALR(1) parsers, i.e. parsers that make decisions based on one look-ahead token. Because Bison has no means for extending the look-ahead, we are left with two possibilities: either restricting the

language accepted by the parser or modifying the lexical analyzer so that it will perform a look-ahead and distinguish the ambiguous cases from each other by returning different tokens to the parser. We chose the latter approach.

We found that in the worst case the SDL grammar requires a look-ahead of three tokens, two of which can be names or character strings. The lexical analyzer will need to keep these strings in its memory and return them to the parser later, when the scanner is called again by the parser. In this way, the look-ahead implemented in the scanner is totally invisible to the parser.

For returning the first of two look-ahead token names to the parser, we will introduce another dummy start condition, ret2. When the scanner has buffered only one look-ahead string, it can be returned by the start condition ret introduced earlier.

3.2.1 1-Token Look-Ahead

Two Meanings of the `operators` Keyword

The SDL grammar rules in Figure 3.2 cause problems that can only be solved by distinguishing the two meanings of the `operators` in the scanner. These two rules seem totally unrelated. What on earth could cause a conflict? The answer lies in the surrounding grammar rules, which are presented in Figure 3.3.

The alert reader will notice that in ⟨*partial type definition*⟩, the non-terminal symbol ⟨*properties expression*⟩, whose expansion can begin with ⟨*operator list*⟩, can immediately follow the non-terminal symbol ⟨*extended properties*⟩ and thus ⟨*inheritance rule*⟩. When the parser reads an `OPERATORS` token while processing an ⟨*inheritance rule*⟩, it does not know whether to shift or to reduce.

The conflict could be resolved by removing the meaningless [`OPERATORS`] from ⟨*inheritance rule*⟩, but it would make the parser incompatible with SDL, which we do not want. Instead, we must modify the lexical analyzer so that it returns a different token for the `operators` keyword, if the keyword is followed by a left parenthesis or by the `all` keyword, which cannot occur in the beginning of ⟨*operator signature*⟩. This can be achieved using two start conditions.

The first start condition operators1 is called whenever ret is about to return the `OPERATORS` token. It skips any whitespace and comments. If it encounters an alphanumeric string, it will switch to operators2 and call word2. Otherwise, the

⟨*operator list*⟩ := `OPERATORS` ⟨*operator signature*⟩
(⟨*end*⟩ ⟨*operator signature*⟩)* [⟨*end*⟩]

⟨*inheritance rule*⟩ := `INHERITS` ⟨*type expression*⟩ [⟨*literal renaming*⟩]
[[`OPERATORS`] (`ALL` | `(` ⟨*inheritance list*⟩ `)`) [⟨*end*⟩]]
[`ADDING`]

Figure 3.2: SDL grammar rules containing the `OPERATORS` token

⟨*partial type definition*⟩ := NEWTYPE ⟨*name*⟩
[⟨*formal context parameters*⟩]
[⟨*extended properties*⟩]
⟨*properties expression*⟩
ENDNEWTYPE [⟨*name*⟩]
⟨*extended properties*⟩ := ⟨*inheritance rule*⟩
⟨*properties expression*⟩ := ⟨*operators*⟩
(⟨*internal properties*⟩|⟨*external properties*⟩)
[⟨*default initialization*⟩]
⟨*operators*⟩ := [⟨*literal list*⟩][⟨*operator list*⟩]

Figure 3.3: Grammar rules referring to the rules in Figure 3.2

⟨*path item*⟩ := SYSTEM ⟨*name*⟩
| SYSTEM TYPE ⟨*name*⟩
| TYPE ⟨*name*⟩

Figure 3.4: Partial rules for SDL path items

scanner will return OPERATORS_, or OPERATORS if the next character is a left parenthesis.

In the second start condition, the scanner will check the name read by word2. If it is the `all` keyword, OPERATORS is returned to the parser. Otherwise the parser will get an OPERATORS_ token. In either case, the scanner will switch to ret, which will return the look-ahead token.

Converting Keyword Pairs to one Token

The SDL grammar contains troublesome rules like the ones presented in Figure 3.4. Together with some other grammar rules they cause many shift-reduce conflicts. The problem can be addressed by returning a single, distinct token for the keyword pair `system type`. This is achieved using two scanner start conditions.

When the scanner encounters the keyword `system`, it will switch to the start condition system, which processes whitespace and comments in the normal way. If there are letters in the input, the scanner will switch to the start condition system2 and call word2. Otherwise it will return to the calling start condition and return the token SYSTEM to the parser.

In the start condition system2, the name read by word2 will be checked. If it is the keyword `type`, the token SYSTEMTYPE will be returned, and the scanner will

⟨*literal list*⟩ := `LITERALS` ⟨*literal signature*⟩
(`,` ⟨*literal signature*⟩)* [⟨*end*⟩]
⟨*literal signature*⟩ := ⟨*name*⟩
| ⟨*extended literal name*⟩
⟨*extended literal name*⟩ := ⟨*character string*⟩
| `NAMECLASS` ⟨*regular expression*⟩
⟨*literal renaming*⟩ := `LITERALS` ⟨*literal rename list*⟩
⟨*literal rename list*⟩ := ⟨*literal rename pair*⟩
(`,` ⟨*literal rename pair*⟩)*
⟨*literal rename pair*⟩ := ⟨*literal rename signature*⟩ `=`
⟨*literal rename signature*⟩
⟨*literal rename signature*⟩ := ⟨*name*⟩
| ⟨*character string*⟩
⟨*literal equation*⟩ := `FORALL` ⟨*name list*⟩
`IN` ⟨*extended sort*⟩
`LITERALS (` ⟨*literal axioms*⟩
(⟨*end*⟩ ⟨*literal axioms*⟩)* [⟨*end*⟩] `)`

Figure 3.5: SDL grammar rules related with the `literals` keyword

pop from the stack the start condition that called system. Otherwise, the parser will receive the token `SYSTEM`, and the scanner will switch to ret, which will return the look-ahead token (name or malformed name) to the parser on the next call to the scanner.

Similar start conditions exist for the keyword pairs `block type`, `process type` and `service type`. Also the start condition ret2, which is used to return the first of two look-ahead tokens to the parser, will convert the keyword pairs to single tokens.

3.2.2 2-Token Look-Ahead

Two Tokens for `literals`

The `literals` keyword occurs in three grammar rules (Figure 3.5). Similarly as with the `operators` keyword (see Section 3.2.1), conflicts occur if the same token is used in all rules. They can be resolved by using a different `LITERALS_` token in ⟨*literal list*⟩.

By looking at the rules, one can notice that the `LITERALS_` token should be returned if the `literals` keyword is followed by the `nameclass` keyword or by a ⟨*name*⟩ or ⟨*character string*⟩ not followed by an equal sign. This can be achieved using only two start conditions, both of which will skip whitespace and lexical

⟨*operator definition*⟩ := ⟨*package reference clause*⟩*
`OPERATOR` ⟨*identifier*⟩ ⟨*end*⟩
...
⟨*identifier*⟩ := [⟨*qualifier*⟩] ⟨*name*⟩
⟨*qualifier*⟩ := ⟨*path item*⟩ (`/` ⟨*path item*⟩)*
⟨*path item*⟩ := `OPERATOR` (⟨*name*⟩[`!`] | ⟨*quoted operator*⟩)
⟨*generator parameter*⟩ := `OPERATOR` ⟨*name list*⟩
⟨*textual operator reference*⟩ := `OPERATOR` ⟨*name*⟩
[[⟨*formal parameters*⟩] ⟨*operator result*⟩]
`REFERENCED` ⟨*end*⟩
⟨*formal parameters*⟩ := `FPAR` ⟨*parameters of sort*⟩
(`,` ⟨*parameters of sort*⟩)*
⟨*operator result*⟩ := `RETURNS` [⟨*name*⟩] ⟨*identifier*⟩

Figure 3.6: Some SDL grammar rules involving the `operator` keyword

comments.

The first start condition, literals1, is called by ret when it is about to return a `LITERALS` token. When encountering a name or a character string, it will switch to literals2 and call word1, word2 or sdlstring to read in the string. Any other input will make it return `LITERALS` to the parser, put back the input and return to the start condition that called ret.

The second start condition, literals2, makes the final decision. If the first look-ahead token was the keyword `nameclass` or if the next non-whitespace character is not an equal sign, a `LITERALS_` token will be returned instead of `LITERALS`. In either case, the scanner will put back the look-ahead character and switch to ret, which will return the first look-ahead token.

3.2.3 3-Token Look-Ahead

The `operator` Keyword

The grammar rules presented in Figure 3.6 are responsible for numerous shift-reduce conflicts. The conflicts can be resolved by using a different `OPERATOR_` token for the first ⟨*path item*⟩ in ⟨*qualifier*⟩.

Now we need nothing less than three new start conditions, the first two of which will skip whitespace and comments. ret calls operator1 when it is about to return `OPERATOR` to the parser. In this start condition, the scanner will advance to operator2 and call word1, word2 or op1 when it encounters anything that looks like a name or a quoted operator. On any other input, the scanner will return to the caller of ret and return an `OPERATOR` token to the parser.

The second start condition, operator2, can make some further decisions. It

will extend the look-ahead name with any exclamation points it encounters (and convert the look-ahead token to *OP_NAME* or *ERR_OP_NAME*). A dot (which starts a ⟨*name*⟩ but not any keyword) makes the scanner switch to ret for returning the look-ahead token and return *OPERATOR* or *OPERATOR_* to the parser. *OPERATOR* will be returned if the look-ahead token can start a ⟨*path item*⟩.

When an alphanumeric string is encountered in operator2, the scanner will check whether the first look-ahead token starts a ⟨*path item*⟩. If it does, the scanner puts back the second string, switches to ret for returning the look-ahead string and returns *OPERATOR*. Otherwise the scanner switches to operator3 and calls word2 for reading in the second look-ahead token.

If any other input is encountered, the scanner will check if the character is a slash (/). If it is, an *OPERATOR_* token can be returned. Otherwise the parser will get an *OPERATOR* token after the scanner has put back the character and switched to ret.

The third start condition, operator3, will check the second look-ahead token. If it is one of the keywords `fpar`, `referenced` or `returns` (see Figure 3.6), *OPERATOR* will be returned. Otherwise the scanner gathers that it has read an ⟨*identifier*⟩ with a ⟨*qualifier*⟩ consisting of one ⟨*path item*⟩ and returns the token *OPERATOR_*. In either case, the scanner will switch to ret2 for returning the two look-ahead tokens.

The `returns` Keyword

The rule for ⟨*operator result*⟩ in Figure 3.6 is self-conflicting. When given the input *RETURNS* ⟨*name*⟩, the parser has two choices with regard to this rule. It could shift the next token, assuming that the optional ⟨*name*⟩ is present, or it could reduce, assuming that the ⟨*name*⟩ in the input read so far actually belongs to the ⟨*identifier*⟩ part of the rule.

This conflict was originally solved with a 3-token look-ahead. Luckily there is a much cleaner solution. When the optional ⟨*name*⟩ is changed to ⟨*identifier*⟩, there is no conflict. A semantic action will ensure that if present, the optional ⟨*identifier*⟩ is a ⟨*name*⟩.

Chapter 4

User-Friendliness: Error Reporting

Error messages from a compiler are something that are disliked and appreciated at the same time. The user may first be annoyed by an error report he receives for his program, but he will appreciate it if the report is clear and assists him in correcting the errors. The worst thing a compiler can do with erroneous input is to ignore the errors and produce something unpredictable that seems to be correct at the first glance.

It is customary to distinguish between two kinds of errors [1, p. 161]. *Syntactic errors* are violations against the grammar incorporated into the parser. In the case of LALR(1) languages, they are caused by sequences of symbols that cannot be reduced by any rule in the grammar. The lexical analyzer and the parser must deal with this kind of errors and try to recover from them, so that the compiler does not need to stop at the first error.

Semantic errors, on the other hand, are expressions that violate the language definition proper even though they are accepted by the parser's typically somewhat liberal grammar.[1] For instance, referring to previously undefined variables in a programming language or calling an undefined subroutine can be a semantic error. Such errors are mainly detected by later compiler stages that operate on the syntax tree level, which is discussed in Chapter 5.

4.1 Error Handling in Bison

By default, when a Bison-generated parser detects a grammar violation in the input, it will report "parse error" and stop parsing, ignoring the rest of the input. This is unacceptable for a compiler, since the user expects to receive reports for all errors contained in the program, so that he needs not run the compiler after correcting each individual error.

Yacc and Bison have a special symbol ⟨*error*⟩ that can be used for error recovery. Whenever a syntax error occurs, an ⟨*error*⟩ token will be generated by the parser. *Error recovery rules*, grammar rules with an ⟨*error*⟩ symbol on their right-hand-side, define a kind of synchronization points. With their help, the parser can resume normal operation. If the parse stack does not match any error recovery

[1] Note that generally the grammar is context-free but the programming language is not.

$$
\begin{array}{rcl}
\langle \textit{item list} \rangle & := & \langle \textit{item} \rangle \\
& | & \langle \textit{item list} \rangle \texttt{ , } \langle \textit{item} \rangle \\
& | & \langle \textit{error} \rangle \\
& | & \langle \textit{item list} \rangle \texttt{ , } \langle \textit{error} \rangle
\end{array}
$$

Figure 4.1: Error recovery rules for a comma-delimited list

rule, states and symbols from it will be discarded until an error recovery rule can be applied.

For a compiler it is often desirable to recover from syntax errors as soon as possible. To achieve this, there must be ⟨*error*⟩ rules at a very deep level in the grammar. Even the lexical analyzer can correct some errors. Malformed ⟨*name*⟩s and ⟨*quoted operator*⟩s are detected by the scanner, which converts them to special tokens, which are taken care of by the parser.

4.1.1 Sequences of Symbols

Programming languages contain many list constructs. For instance, variable definitions and function parameters can be seen as lists (usually delimited by a comma or by some other character). Introducing error recovery rules for these constructs is essential. The most straightforward way of doing so is adding the ⟨*error*⟩ tokens as alternatives to the list items, as shown in Figure 4.1. The same could be achieved by augmenting the grammar with the rule

$$\langle \textit{item} \rangle \quad := \quad \langle \textit{error} \rangle .$$

4.1.2 Frequently Used Symbols

Fast synchronization on errors can be achieved by introducing error recovery rules at the lowest possible level. For instance, the SDL grammar uses ⟨*identifier*⟩ in very many places. Augmenting the rule for ⟨*identifier*⟩ with an ⟨*error*⟩ token seems to make it possible to synchronize on errors almost everywhere.

Unfortunately, the ⟨*identifier*⟩ symbol is used in so many different contexts in the SDL grammar that allowing ⟨*error*⟩ wherever ⟨*identifier*⟩ is allowed would introduce too many conflicts to the grammar. Therefore, it makes sense to define another non-terminal symbol,

$$
\begin{array}{rcl}
\langle \textit{identifier}' \rangle & := & \langle \textit{identifier} \rangle \\
& | & \langle \textit{error} \rangle
\end{array}
$$

which is used instead of ⟨*identifier*⟩ wherever doing so does not introduce conflicts. Similar rules are defined for many other frequently used symbols.

4.1.3 Fall-Back Rules

Sometimes the parser comes very badly out of synchronization with the input. Without any so-called fall-back rules, the parser could be forced to discard everything following a bad error. Two fall-back rules in our SDL parser allow the parser to attempt synchronization after discarding anything up to a semicolon, which is contained in ⟨*end*⟩:

$$\begin{aligned} \langle \textit{entity in entity} \rangle &:= \langle \textit{error} \rangle [\langle \textit{end} \rangle] \\ \langle \textit{transition} \rangle &:= \langle \textit{error} \rangle \langle \textit{end} \rangle \end{aligned}$$

In addition, some rules define ⟨*error*⟩ alternatives for a sequence of symbols, like the following:

$$\begin{aligned} \langle \textit{signal list definition} \rangle &:= \texttt{SIGNALLIST}\ \langle \textit{name} \rangle = \langle \textit{signal list} \rangle\ \langle \textit{end} \rangle \\ &\mid\ \texttt{SIGNALLIST}\ \langle \textit{error} \rangle\ \langle \textit{end} \rangle \end{aligned}$$

Unterminated lexical comments and character strings are handled by the end-of-file rules of the lexical analyzer. The scanner does not try to make any guesses where the comment or character string should end; it merely treats the rest of the file as one comment or string and issues an appropriate error message.

4.1.4 Shift-Reduce Conflicts Involving the ⟨*error*⟩ Token

As a result of introducing error recovery rules to the grammar, the SDL parser contains some shift-reduce conflicts where it will decide to shift the ⟨*error*⟩ token instead of reducing. Since these conflicts do not affect the parsing of correct input, we did not try hard to remove them. Shifting the ⟨*error*⟩ token means that the parser may need to discard more input to recover from an error than it would if there was no conflict. On the other hand, the error recovery rule causing the shift-reduce conflict may make the parser synchronize faster in some cases.

4.2 Relaxing the Grammar

In the previous section we saw that augmenting the grammar with rules containing ⟨*error*⟩ tokens allows the parser to recover from errors in the input. The goal was to obtain as many parse errors from erroneous input as possible; in the best case one parse error for each error in the input. But often we want to proceed even in the seemingly opposite direction and reduce the number of parse errors generated, without touching the rules containing ⟨*error*⟩ symbols. The reason is that a descriptive error message such as “name expected”, “missing comma” or “extraneous comma” satisfies the user more than a generic “parse error”. Such error messages can be implemented by extending the grammar and by adding the error messages to the semantic actions of these “extraneous” rules.

Improving error recovery is a kind of art. The compiler cannot recognize all kinds of errors and adapt to them: idiots are clever, they say, and users may always introduce errors that the compiler does not handle very well. In the SDL parser we tried to relax the grammar as much as possible without introducing conflicts.

4.2.1 Tolerating Too Generic Constructs

Sequences of Symbols Instead of Symbols

Some places of the SDL grammar can be relaxed so that lists or sequences of symbols are tolerated where only one symbol is allowed. Returning to the example illustrated in Figure 4.1, a grammar rule

$$\langle item' \rangle \quad := \quad \langle item\ list \rangle$$

that verifies that the $\langle item\ list \rangle$ only has one component and issues an error message if not, can be used to replace $\langle item \rangle$ in the grammar wherever doing so does not introduce conflicts.

Single Symbols

Programming languages often have many overlapping non-terminal symbol definitions. For instance, there may be a distinction between *lvalues* (left-hand-side values of assignments) and other expressions. The grammar usually remains parsable even if any expression is tolerated on both sides of the assignment. Semantic analysis can determine later whether the assignment makes sense.

In the SDL grammar it is sometimes possible to tolerate expressions or even expression lists where only a single $\langle identifier \rangle$ or $\langle name \rangle$ is allowed. Sometimes a missing symbol can be tolerated without causing any conflicts.

4.2.2 Tolerating Extraneous Symbols

Repetition of Optional Symbols

The SDL grammar rules contain many optional symbols, that is, there are rules which allow the empty sequence or one occurrence of a symbol. The grammar usually will not become any more difficult for the parser if multiple occurrences of the symbol are allowed. Semantic actions can be used for telling the user that the optional construct can be specified at most once.

Other Symbols

A draft of [27, Annex D], which defines the built-in SDL data types in terms of SDL, contained lots of errors. For instance, there were extraneous commas, and at one place there was an extraneous right parenthesis. When testing the parser, one error that is present in the published version of [27, Annex D] was found. In the definition of the built-in `Integer` type, there is an axiom that is not allowed by the SDL grammar:[2]

```
type Integer - 0 == 0;
```

[2] The correct form would be "`type Integer "-" (0) == 0;`."

$$
\begin{array}{rcl}
\langle \textit{no end} \rangle & := & [\langle \textit{end} \rangle] \\
\langle \textit{no qualifier} \rangle & := & [\langle \textit{qualifier} \rangle] \\
\langle \textit{no commas} \rangle & := & \varepsilon \\
 & | & \texttt{,}\ \langle \textit{no commas} \rangle \\
\langle \textit{no rparens} \rangle & := & \varepsilon \\
 & | & \texttt{)}\ \langle \textit{no end} \rangle\ \langle \textit{no rparens} \rangle \\
\langle \textit{no op} \rangle & := & \varepsilon \\
 & | & \langle \textit{no op} \rangle\ \langle \textit{op} \rangle
\end{array}
$$

Figure 4.2: The “no” rules of the SDL parser

These errors were the inspiration for some new non-terminal symbols in the Bison definition file, presented in Figure 4.2. In the last rule $\langle \textit{no op} \rangle$, which is used in expressions after binary operators like this,

$$\langle \textit{sub expression} \rangle \quad := \quad \langle \textit{sub expression} \rangle + \langle \textit{no op} \rangle\ \langle \textit{sub expression} \rangle,$$

the non-terminal symbol $\langle \textit{op} \rangle$ expands to any operator except the unary operators `NOT` and `-`, which would introduce conflicts.

The grammar of the SDL parser was augmented with these “no” non-terminals in several places where doing so made sense and did not introduce conflicts. In order to parse the erroneous axiom for the `Integer` type, the rules for unary operators in $\langle \textit{term} \rangle$s were augmented as follows:

$$
\begin{array}{rcl}
\langle \textit{term} \rangle & := & \langle \textit{no qualifier} \rangle\ \texttt{-}\ \langle \textit{no op} \rangle\ \langle \textit{term} \rangle \\
 & | & \langle \textit{no qualifier} \rangle\ \texttt{NOT}\ \langle \textit{no op} \rangle\ \langle \textit{term} \rangle
\end{array}
$$

The semantic action of $\langle \textit{no qualifier} \rangle$ will issue an error message “Qualifier not allowed here” for the erroneous definition in [27, Annex D].

Chapter 5

Storing the Input: The Syntax Tree

A parser that does not store the parsed data is of no use, unless the goal is nothing more than simple syntax checking. A compiler must store everything that is relevant in further processing of the input. How should the input be stored? [1, p. 49] describes a translator for simple expressions and discusses the difference between the *parse tree* (the syntactic structure of the language) and the data structures needed by the compiler:

> A useful starting point for thinking about the translation of an input string is an *abstract syntax tree* in which each node represents an operator and the children of the node represent the operands. By contrast, a parse tree is called a *concrete syntax tree*, and the underlying grammar is called a *concrete syntax* for the language. Abstract syntax trees, or simply *syntax trees*, differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees.

For simple expressions, the difference between parse tree and syntax tree is subtle. In more complex languages, the syntax tree usually looks very different from the parse tree. For easing compiler writers' work, the standard [27] has an "Abstract grammar" section for each major construct of SDL. These sections define the framework of one possible syntax tree for SDL.[1]

5.1 Class Hierarchy

Since our SDL compiler is implemented in C++, the most natural way of defining the syntax tree is in terms of C++ classes. Each syntax tree node is a C++ object.

There are two kinds of relations between the syntax tree nodes in our class hierarchy. Nodes can consist of other nodes (or have references to them), or they can be derived from a common base node or from each other.

In our SDL compiler, syntax tree node classes are divided into four major groups: ENTITY (described in Section 5.1.1), EXPRESSION (Section 5.1.2), SORT

[1] The syntax tree is an internal data structure of the compiler, and it can be constructed in many ways.

(Section 5.1.3) and STATE (Section 5.1.4). These classes make use of some utility classes that are discussed in Section 5.1.5, e.g. IDENTIFIER and PATHITEM for storing ⟨*identifier*⟩ data. We use THIS typeface for class names.

5.1.1 ENTITY

Classes derived from the ENTITY class span the frame of the syntax tree in our SDL compiler. The root of the tree is a SPECIFICATION object. Each ENTITY has its own name space and scope of visibility, and apart from other ENTITY objects, different ENTITY objects can contain different definitions (objects not derived from the ENTITY class). For instance, a PROCESS can contain an AUTOMATON, and a DATA object (which corresponds to ⟨*data definition*⟩ in the grammar) contains a SORT.

SPECIFICATION

The SPECIFICATION class is in a special position. It is only instantiated once in the SDL compiler, and all SDL entities defined at the top level will be stored in the only instantiated object of SPECIFICATION. Thus, SPECIFICATION is the root of the syntax tree and comprises all the relevant input fed to the compiler.

PACKAGE

In SDL, entities are usually grouped into packages, which correspond to modules in many other programming languages. Usually each package in an SDL specification is defined in a separate file, but our compiler does not have any restrictions regarding this.

When an ASN.1 parser is integrated with the compiler, the ASN.1 modules will be converted to PACKAGE objects.

SYSTEM

According to [27], an SDL specification consists of one system and a set of other definitions. The System entity in SDL can be seen as a special kind of package. The SYSTEM class corresponds to the ⟨*textual system definition*⟩ in the SDL grammar. Like PACKAGE, it is only a wrapper for other definitions.

SYSTEMTYPE and SYSTEMINST

SDL is said to be an object-oriented language [9], although it lacks many of the properties one excepts from an object-oriented language, such as polymorphism, multiple inheritance, or packaging data structures and related algorithms into classes.

With some imagination, one can see similarity between types in SDL and classes in real object-oriented languages.[2] In object-oriented languages, objects are instantiated from classes. In SDL, entities are instantiated from entity types.

Entity types are more like templates or macros than classes. They are parameterized entities. The parameters will be fixed when the entity type is instantiated.

SYSTEMTYPE is otherwise the same as SYSTEM, but it has formal parameters (FPAR) that will be supplied in the TYPEEXPRESSION of a SYSTEMINST. A system type definition can also be based on another system type; the optional ⟨*specialization*⟩ is stored in the TYPEEXPRESSION member of SYSTEMTYPE.

Since an SDL specification may only contain one system definition, it is an error for a SPECIFICATION to contain more than one SYSTEM or SYSTEMINST.

SYSTEMTYPEREF

In SDL, entities are defined inside other entities, and they are normally "visible" only to entities that are defined in the same scope. For instance, if a system type is defined in one package, it cannot be directly used by an entity defined in some other package. The ⟨*textual system type reference*⟩ in SDL addresses this problem, and it is mapped to the SYSTEMTYPEREF class in the SDL compiler. Systems need not be referenced, since there is only one system in a SDL specification.

BLOCK

Blocks in SDL are yet another container for other entities. Unlike SYSTEM and PACKAGE, BLOCK allows the definition of processes and thus the definition of behavior. Blocks are connected to each other through CHANNEL objects, whose names are stored in CONNECTION objects.

BLOCKTYPE and BLOCKINST

The instantiation mechanism associated with BLOCK is similar to the one associated with SYSTEM. As a new thing, BLOCKTYPE encompasses GATE definitions, which are instantiated to CONNECTION objects. In addition, block types have virtuality constraints and options, which control the inheritance.

BLOCKREF and BLOCKTYPEREF

The BLOCKREF and BLOCKTYPEREF classes are for references to BLOCK and BLOCKTYPE objects, respectively.

PROCESS

Processes are the components in SDL that define the behavior of the system. They may have creation-time parameters, a set of input signals, connections (signal routes) to other entities and an enhanced finite state automaton associated with

[2] The word "type" has no other meaning in SDL; abstract data types are referred to with the term "sort."

them.[3] In a process declaration one can also specify the initial number of process instances that will be created when the system is initialized, and the maximum number of instances of the process that are allowed to exist simultaneously.

PROCESSTYPE and PROCESSINST

Processes can be instantiated from process types just like blocks can be instantiated from block types.[4] The initial and maximum number of process instances and formal parameters are specified in the PROCESSINST object; everything else is defined in the corresponding PROCESSTYPE object.

PROCESSREF and PROCESSTYPEREF

The PROCESSREF and PROCESSTYPEREF are for references to PROCESS and PROCESSTYPE objects, respectively.

PROCEDURE, PROCEDUREREF, REMOTEPROCEDURE and IMPORTEDPROCEDURE

There are two kinds of procedures in SDL: local procedures and remote procedures. Local procedures are executed by the calling process, while remote procedures are executed by a different process. The PROCEDURE class encompasses local procedure definitions, and PROCEDUREREF is used for storing references to PROCEDURE objects.

Procedures have an interface defined in terms of parameter sorts and return sort. The actions of a procedure are defined in the AUTOMATON object contained in it.

Remote procedures in SDL are otherwise defined like local procedures, but in the preamble, the procedure is given a remote procedure identifier which identifies the REMOTEPROCEDURE entity that serves remote calls to the procedure. Callers of the remote procedure need an IMPORTEDPROCEDURE entity that declares the calling interface of the procedure.

SERVICE

SDL processes can be partitioned into services. Each service must have a disjoint set of input signals, and only one service of a process is executed at a time. Each SERVICE class has an optional set of input signals and an AUTOMATON object associated with it.

SERVICETYPE, SERVICEINST, SERVICEREF and SERVICETYPEREF

Services and service types can be referenced and instantiated just like procedures and blocks.

[3] Processes divided into services may effectively contain more than one automaton.

[4] There are two different kinds of instantiation here: the compile-time template instantiation from PROCESSTYPE and the creation of run-time processes from PROCESS or PROCESSINST.

SIGNAL and SIGNALS

Signals define the messages passed between processes. There are two classes related to them. SIGNAL stores a single ⟨*signal definition item*⟩, and SIGNALS encapsulates a SIGNALLIST object, which corresponds to a ⟨*signal list*⟩ in the SDL grammar.

DATA

The DATA class corresponds to the ⟨*data definition*⟩ symbol in the SDL grammar. An object of this class contains a reference to the actual data type definition, which is a SORT object, to the initialization expression for variables of this sort, and to any constraints set for variables of the sort, among other things.

OPERATOR and OPERATORREF

It is possible to define operators for abstract data types. Operators are more restricted than local procedures, since they can only be defined in a ⟨*data definition*⟩ and may only contain variable and sort definitions and an automaton. The AUTOMATON associated with an OPERATOR may not have any persistent state information, and it cannot communicate with other entities.

CHANNEL

Blocks communicate through channels and processes through signal routes. Both definitions are stored in CHANNEL objects. Unlike signal routes, channels may have internal behavior defined in terms of SUBSTRUCTURE.

SUBSTRUCTURE and SUBSTRUCTUREREF

SDL supports modularity and hierarchical designs by allowing the partitioning of blocks into sub-blocks and channels, which in turn can be partitioned into subchannels and blocks. The SUBSTRUCTURE class is used for storing both block and channel partitions.

TIMER

Timers are used in protocols e.g. for preventing potential deadlocks or for detecting dead communication links. In SDL, timers are either inactive or active, and active timers have a time and optional parameters associated with them. If the time associated with a timer elapses before a new time is set for the timer or before the timer is made inactive, the owner of the timer will be sent a timer signal, which will carry the timer name and optional parameters that were specified at the time the timer was last set.

A TIMER object contains a list of parameter sort identifiers and an expression specifying the default duration.

SELECT

SDL has very limited support for conditional compilation. SELECT is one of the few constructs available for that purpose. A ⟨*select definition*⟩ in the SDL grammar consists of a boolean expression and of a set of entity definitions. If the expression is true, the entities defined inside the ⟨*select definition*⟩ will be treated as if they were defined in the surrounding entity. Otherwise the definitions are discarded from the specification.

5.1.2 EXPRESSION

Expressions are very central in any programming language. In our SDL compiler, all expression nodes are derived from the abstract base class EXPRESSION. Some of the expression types, implemented as EXPRESSION subclasses, are present in virtually any programming language, some are SDL specific.

CHARSTRINGEXPRESSION and LITERALEXPRESSION

The simplest leaves of SDL expression trees are character strings and literals (identifiers). In SDL, also numbers are treated as names or identifiers. CHARSTRINGEXPRESSION is associated with a string and an optional qualifier; LITERALEXPRESSION encapsulates an IDENTIFIER object, which contains the literal name and the optional qualifier. A LITERALEXPRESSION can be almost anything, for instance a numeric constant, a literal of an enumerated sort, or a reference to a variable.

NOPEXPRESSION

In SDL expressions, there are some reserved words that can be viewed as operators with no arguments. These words are handled by NOPEXPRESSION objects.

UNOPEXPRESSION

SDL has two unary operators, the negation operator "`-`" and the logical "`not`" operator. The UNOPEXPRESSION class encapsulates a unary operator and an EXPRESSION associated with it.

BINOPEXPRESSION

There are plenty of binary operators in SDL, that is, operators that have a left-hand-side expression and a right-hand-side expression. Binary operator expressions are handled by the BINOPEXPRESSION class.

IFTHENELSEEXPRESSION

The ternary if-then-else operator familiar from the C programming language is available in SDL, and IFTHENELSEEXPRESSION objects store the if-then-else nodes of the expression tree.

STRUCTCOMPONENT and STRUCTEXPRESSION

A tuple of variables is called `struct` in SDL. A `struct` has named components. Individual components of a `struct` are accessed by name. The STRUCTCOMPONENT class handles such accesses.

All components of the whole `struct` can be initialized at once with a special notation. STRUCTEXPRESSION encapsulates the expressions used for initializing the components.

STRUCTOROPERATOR and OPERATORAPPLICATION

SDL has two syntaxes for accessing `struct` components, and one of them conflicts with OPERATORAPPLICATION, which is used when a user-defined operator is applied to, or "called with" some argument expressions. The STRUCTOROPERATOR class is used for dealing with these definitions. Also accesses to array elements are catched by this class. All STRUCTOROPERATOR objects will be converted to other objects in later compiler stages.

PROCEDURECALL and REMOTEPROCEDURECALL

Local and remote procedure calls are semantically close to user-defined operator applications, but since procedures and user-defined operators have separate namespaces, it is easiest to write separate classes for procedure calls: PROCEDURECALL for local and REMOTEPROCEDURECALL for remote calls. Both classes encapsulate the procedure identifier and the actual argument expression list. For REMOTEPROCEDURECALL, also an expression identifying the call destination is recorded.

IMPORTEXPRESSION

When a remote variable is imported, its value is copied from the process holding the master copy of the variable. IMPORTEXPRESSION records the variable to be imported, the expression identifying the process, and an optional message path which should be used when importing the variable.

VIEWEXPRESSION

The deprecated `view`/`revealed` concept in SDL effectively lets processes share global variables. Processes viewing a global variable cannot modify its value; only the process that declared the variable can. VIEWEXPRESSION records the variable identifier and an expression identifying the process whose master copy of the variable is to be viewed.

ACTIVEEXPRESSION

An ACTIVEEXPRESSION is a boolean expression that enquires whether a timer instance is active. Its parameters are the timer identifier and the optional timer parameters.

Table 5.1: Built-in leaf sorts in SDL

`Boolean`	Boolean truth values and operators
`Character`	A single 7-bit International Alphabet 5 character
`Charstring`	`String(Character,'')`
`Integer`	Arbitrary precision integer number
`Natural`	Non-negative arbitrary precision integer number
`Real`	Arbitrary precision floating point number
`PId`	Process identifier
`Duration`	A `Real` value indicating duration of a time interval
`Time`	A `Real` value indicating absolute time

ANYEXPRESSION

The ANYEXPRESSION is associated with the ⟨*anyvalue expression*⟩ in the SDL grammar. It denotes an unspecified value of a sort, i.e. it will evaluate to any value of the specified sort.

RANGE

In SDL, it is possible to define constraints for values of a data type. Since these constraints are defined in terms of expressions, it is natural to use an EXPRESSION-derived class for them. The abstract RANGE class and its subclasses handle the value constraints in SDL.

5.1.3 SORT

The abstract data type, or sort system in SDL has some peculiarities. First of all, SDL supports axiomatic definition of data types. It seems that axioms are only used for defining the built-in data types in [27, Annex D]; it would be extremely difficult to write a compiler that would interpret the axioms, check that they are solid and complete, and finally translate the axioms to an efficient implementation of the data type and its operators. Our compiler parses axiomatic definitions properly, but it ignores them and issues error messages saying that axioms are not supported.

There are two kinds of sorts in SDL: simple sorts, or leaf sorts, as presented in Table 5.1, and structured sorts, which consist of other sorts. SDL does not have anything corresponding to the `union` type of C. Our compiler supports `choice` with a `struct`-like syntax as a non-standard addition inspired by ASN.1. The structured sorts are as follows:

`Array(Index,Itemsort)`

An array of `Itemsort` indexed by `Index` sort. As opposed to many other programming languages, `Index` sort need not be an integer-like sort; SDL arrays can be indexed e.g. by `Charstring` or by a structured sort.

`struct`
: A `struct` sort consists of named components of any sort. A `struct` value defines values of all the `struct` components.

`choice`
: The `choice` sort is like `struct`, but only one of the components is active at a time. Thus, a `choice` value must actually be a value of one of the `choice` component sorts.

`Powerset(Itemsort)`
: A power set of `Itemsort`. `Powerset` could be implemented in terms of `Array` as `Array(Itemsort,Boolean)` augmented with some operators.

`String(Itemsort,Emptystring)`
: String of `Itemsort`; a kind of `Natural`-indexed array. `Emptystring` specifies the literal that will be used for denoting the empty string.

`syntype`
: Synonym type; a sort alias. The sort defined by `syntype` inherits the data structure from the original sort and possibly augments it with some literals, constraints and operators.

Simple Sorts

The classes defining simple sorts do not have any member objects; everything is inherited from the base class SORT. The SORT class has a reference to the DATA entity containing the sort definition, and it holds the literals defined for the sort. In addition, the base class SORT keeps track of the operators defined for the sort.

ARRAYSORT

The ARRAYSORT class corresponds to the `Array` generator in SDL. It has two members: the index sort and the item sort.

STRUCTSORT and CHOICESORT

The two classes STRUCTSORT and CHOICESORT implement the structured sorts `struct` and `choice`, respectively. The component sorts are stored in a COMPONENTLIST.

POWERSETSORT

For now, the POWERSETSORT class only stores the power set item sort.

STRINGSORT

The STRINGSORT class handles the `String` generator. The item sort and the empty string literal (name or character string) are stored.

SYNTYPESORT

The SYNTYPESORT class only records the original sort. Added literals and operators are kept track by the SORT base class, and constraints are associated with the DATA entity encapsulating the SYNTYPESORT object.

5.1.4 STATE

Usually, all actions in an imperative programming language take place in *statements*, which form the bodies of functions and procedures. SDL, as a language designed for modeling communicating automata, has the concept of *states* at the function or procedure body level. For instance, a process may have a number of named states, each of which is associated with conditions or input signals. When the process receives an input signal that it is expecting in its current state, it will execute a sequence of statements, the *transition* associated with that input and state.[5]

AUTOMATON

The AUTOMATON class glues the building blocks of extended finite state automata together. It has a reference to the surrounding ENTITY, and it has references to all states and transitions (statement sequences) of the automaton. One statement sequence is in a special position: the (in some cases optional) start transition is executed when the automaton is entered, before waiting for any input. AUTOMATON also keeps track on the names of states and on labels, which can precede any statement and act as entry points.

During parsing, the AUTOMATON class maintains some auxiliary structures that are needed for expanding the short-hand notations in SDL, such as defining the behavior of all states by using the "`state *`" construct.

STATE

SDL automata leave their current state upon receiving a signal, or when a logic condition, specified in terms of a Boolean expression, becomes true. Figure 5.1 represents a simple process with two states, called `Idle` and `Active`. Initially, the process will switch to the `Idle` state, where it is expecting two signals, `Atn` and `Quit`. Upon receiving the former signal, the process will switch to the `Active` state; the `Quit` signal will be saved to be consumed later. In its `Active` state, the process only expects the `Quit` signal, which will cause the termination of the process. The process can also return to the `Idle` state if the ⟨*anyvalue expression*⟩ "`any (Boolean)`" evaluates to `True`.

The abstract STATE class manages the conditions and actions associated with states in SDL automata. The AUTOMATON class keeps track of the state names

[5] The SDL states and transitions are not to be mixed with the places and transitions of Petri Nets [20]; SDL specifications typically have variables, which contain state information, and an SDL transition usually involves communication and other non-atomic tasks.

```
process Simple;
 start;
 nextstate Idle;

 state Idle;
  input Atn;
   nextstate Active;
  save Quit;

 state Active;
  input Quit;
   stop;
  provided any (Boolean);
   nextstate Idle;
endprocess Simple;
```

Figure 5.1: An SDL process containing a simple automaton

and associates each name with (typically multiple) STATE objects. For example, the AUTOMATON object containing the definitions of Figure 5.1 will associate two objects of STATE-derived classes, INPUT and SAVE, with the state name `Idle`.

Signals can be handled in three ways. They can be consumed immediately or saved to be consumed later. The INPUT class handles the consumption of signals, and the SAVE class corresponds to the grammar rule ⟨*save part*⟩. Unexpected signals will be implicitly consumed by the system, without affecting the current state of the automaton.

The third STATE-derived class, CONTINUOUS, is associated with the non-terminal symbol ⟨*continuous signal*⟩ in the SDL grammar. The state can be left whenever the condition EXPRESSION associated with CONTINUOUS evaluates to `True`.

When a STATE object causes the state to be left, the action or SDL transition (chained STATEMENT objects[6]) associated with it will be executed.

ASSIGNMENT

Assignments could also be treated as expressions, but since SDL does not allow assignments inside expressions, we chose to derive ASSIGNMENT from the abstract STATEMENT base class. Both sides of ASSIGNMENT are EXPRESSION objects, and the left-hand-side must evaluate to a variable.[7]

[6] Here we manage a linked list of STATEMENT objects by ourselves, not using the `list` template of the Standard Template Library, since statements need to be manipulated a lot when performing optimizations.

[7] SDL does not allow e.g. IFTHENELSE in the left-hand-side expression, but there is no reason (other than compatibility with [27]) why it should not be allowed.

DECISION

In SDL, there are two constructs that resemble the `switch` construct familiar from the C programming language. Both constructs, `alternative` and `decision`, translate into DECISION objects, which have a condition EXPRESSION and a list of STATEMENT chains associated with EXPRESSIONs.

When the DECISION is executed, the condition will be evaluated, and the action associated with the first EXPRESSION that evaluates to a value equal to the condition will be executed. There is also an optional `else` part whose action will be executed otherwise. The execution will continue from the STATEMENT following the DECISION.

EXPORT

Values of remote variables must be exported explicitly with a statement. The EXPORT class is associated with ⟨*export*⟩ in the grammar.

EXPRESSIONSTATEMENT

In SDL, procedure calls are expressions. In order to be able to call procedures from automata, we need something that glues EXPRESSION and STATEMENT objects together: the EXPRESSIONSTATEMENT class.

LABEL and JOIN

With LABEL, it is possible to define entry points to STATEMENT chains, and the JOIN statement can be used to jump to an entry point.[8] The SDL grammar does not allow multiple labels to a statement, but our implementation does (and issues an appropriate warning message when needed).

OUTPUT

SDL automata act on input signals. Without output from somewhere, there would not be any input. The OUTPUT statement makes it possible to send signals to other processes, or even to the current process itself.

SETTIMER and RESETTIMER

Automata can set timers with SETTIMER and reset (inactivate) them with RESETTIMER statements.

CREATE

In SDL, processes are created with a statement, not with a built-in function such as `fork(2)` in Unix-like system kernels. The CREATE statement in SDL returns

[8] The SDL keyword `join` is called `goto` in most other programming languages.

status information to its caller through built-in variables, which are covered by NOPEXPRESSION.

NEXTSTATE, RETURN and STOP

Statement chains are terminated in three ways. The automaton can switch to a state (NEXTSTATE), it can return to the caller (RETURN), or the process containing the automaton can be terminated (STOP).

5.1.5 Other Classes

The syntax tree classes presented so far enclose many auxiliary objects, such as IDENTIFIER for storing e.g. sort names. These classes are simple and represent very low-level grammar objects.

NAMELIST

In many places of the SDL grammar, there are lists of names: variable names, state names, literal symbol names. The NAMELIST class wraps a `list<char*>` object and ensures that each name on the list is unique.

IDENTIFIER and PATHITEM

SDL has several namespaces and multiple levels of scoping. An ⟨*identifier*⟩ is a ⟨*name*⟩ optionally preceded by a ⟨*qualifier*⟩, which is a list of ⟨*path item*⟩s identifying the SDL entity where the named object is defined. The qualifier can identify a full path from the root of the entity definition tree (starting from the SPECIFICATION level), or the outer entities can be omitted from it, if the entity containing the identifier is defined in their scope.

Path items are stored in PATHITEM objects, and lists of them are collected to PATHITEMLIST objects. An IDENTIFIER object encapsulates a name (a character string) and an optional PATHITEMLIST.

IDENTIFIERANDPARAMETERS

An IDENTIFIERANDPARAMETERS object encapsulates an IDENTIFIER and an EXPRESSIONLIST. It is used for converting a STRUCTOROPERATOREXPRESSION or a LITERALEXPRESSION that is initially parsed as an EXPRESSION. The class is only used during the parsing stage.

NAMESOFSORT and COMPONENT

The NAMESOFSORT class, as its name hints, encapsulates a NAMELIST and an IDENTIFIER identifying a data type (sort). It is mainly used in variable declarations.

When NAMESOFSORT is used while parsing `struct` or `choice` component declarations, it will be converted to a COMPONENTLIST. Each COMPONENT of

the list is assigned a name from the NAMELIST and the sort identified by the IDENTIFIER of the NAMESOFSORT.

DEFINITIONSELECTION

SDL packages may have an interface, which restricts which definitions of the package are accessible by other packages. Also, when importing definitions, one does not have to import all definitions belonging to the interface of a package. The DEFINITIONSELECTION and DEFINITIONSELECTIONLIST classes are used both for specifying the package interface and for listing the imported definitions.

PACKAGEREFERENCE

References to the definitions in an SDL package are handled by the PACKAGEREFERENCE class. It records both the package name and an optional DEFINITIONSELECTIONLIST, which specifies which definitions of the package are to be used.

An SDL entity may need definitions from several packages. Thus, each ENTITY class has an optional PACKAGEREFERENCELIST member.

CONNECTION and GATE

SDL blocks are connected to each other via connections, which may consist of multiple channels, which are defined in CHANNEL entities. When channels are referred to, their identifiers are stored in CONNECTION objects.

Gates are a kind of connection templates, used in block, process and service types. In addition to constraining the other end point (one is connected to the instantiated block, process or service) they constrain the signals allowed. GATE objects store all this information.

FPAR and TYPEEXPRESSION

Templates would be of no use if they had no parameters. In SDL, the parameters of templates (system, block, process and service types) are called *formal parameters*, and their values are bound with *type expressions*. The FPAR and FPARLIST classes are used for formal parameter definitions. TYPEEXPRESSION is very simple: it contains an IDENTIFIER identifying the template (type) and an IDENTIFIERLIST with the actual parameters to be used in the instantiation.

SIGNALLIST

When signals are referred to in an SDL specification, signal lists are used. A signal list contains items consisting of identifiers of signals, timers, remote procedures or variables or identifiers of other signal lists. In our compiler, the signal lists are SIGNALLIST objects encapsulating SIGNALLISTITEM objects.

VARIABLE

Without variables, a programming language would be of very little use. In SDL, variables may have two names associated with them: the local name and an optional exported name, by which remote entities can refer to it. Each variable has a sort associated with it, and in SDL, variables can be initialized with a default expression. In our SDL compiler, all this information is wrapped together in VARIABLE objects.

5.2 Programming Tricks

In a big programming project one is very likely to encounter problems that have rather tricky solutions. Tricks related with the lexical analyzer have been described in Sections 3.1.3 and 3.2. There are also many tricks specific to C and C++, which are useful in any project.

5.2.1 Making Use of the C Preprocessor

C and C++ compilers process their input in two passes. The preprocessor strips off any comments, expands macros and handles include files. The preprocessed input is fed to the rest of the compiler, which performs the actual parsing.

Programs are usually split into modules, which are compiled separately. The compiled modules are linked together to form the executable program. The interface of a module is defined in a header file, which is included into all modules using it.

Another use for include files is defining preprocessor macros. Commonly used macros can be defined in one file, which is included from any module using the macros. In this way, the macro definitions can be stored in one place, which makes maintenance very easy.

There is a third use for include files. An include file can consist of macro invocations. A module will define the macro, include the file, redefine the macro, include the file again, and so on. This trick was needed in some places in our SDL compiler. Its first application was in the ENTITY class.

Each ENTITY object contains a `map` of each ENTITY subclass, and each `map` is handled in a similar way. The program code became very compact by using the macro technique. A file called `allEntities` has an `E()` macro call for each ENTITY subclass. This file is included several times from the `Entity.h` and `Entity.C` files. Adding a new ENTITY subclass to the compiler does not necessarily need any changes in the base class module; only one line needs to be added to the `allEntities` file. However, everything has its price. The code defined in the macros is difficult to debug. On the other hand, the macro to be debugged can be expanded manually, and once it works, it will continue to work for any set of subclasses defined in the `allEntities` file.

5.2.2 The Standard Template Library

The Standard Template Library (STL) is a welcome addition to a C++ programmer's arsenal. It contains C++ class template definitions for practically every basic building block of data structures one can imagine, saving the programmer from reinventing the wheel once again.

STL has its drawbacks. The STL containers can only store fixed-size objects. That is no problem, as long as inheritance is not used. Objects of a derived class typically have more member variables and thus require more storage than objects of the base class, and casting them to the base class would lose information. The only way to store such objects in STL containers is via pointers or references. A pointer always takes the same amount of storage, no matter what kind of an object it points to. Also, it is sometimes important to be able to store null pointers, e.g. meaning that an EXPRESSION in an EXPRESSIONLIST is empty.

Problems with Pointers

Using pointers introduces many problems, and one of them is dynamic memory allocation. Since C++ does not have garbage collection, all dynamically allocated memory should be freed somewhere, as discussed in Section 6.4. When an STL container object is deallocated, its destructor will call the destructor for each stored component. Since pointers do not have any useful destructors associated with them, the objects whose pointers are stored in the STL container object will not be deallocated.

One often suggested solution to the deallocation problem is using a pointer wrapper class. Instead of storing raw pointers, the Standard Template Library container will store pointer wrapper objects. This introduces one more level of indirection and has its own problems. STL containers involve lots of run-time copying due to the way they must be implemented. In a compiler, most of the copying is unnecessary. When a semantic action in the parser appends a syntax tree node to a list, it typically also wants to transfer the ownership of the node to the list, so that the list will take care of deallocating the node when the list itself is being deallocated from the memory. When an object is stored in an STL container, a fresh copy of it will be made, and if the object passed to STL for storing is no longer needed outside the STL container, it must be deallocated. A parser storing syntax tree objects (instead of pointers to them) in STL containers would involve much allocation and copying immediately followed by deallocation.

Copying pointers is simple and efficient while copying objects having a complex, nested structure is not. Unless the pointer wrapper makes use of reference counting, the object contained in it must be copied every time the pointer wrapper object itself is copied. Also, implicit conversions performed by the C++ compiler can be hazardous. If a function expects a pointer wrapper object and the constructor of the wrapper has a single parameter, the pointer, we are juggling with chain-saws. Now if we pass a raw pointer to the function, the C++ compiler will allocate a pointer wrapper object on the stack for us, call the function and deallocate the pointer wrapper. Typically this is not intended, and finding the cause of

the unwanted behavior can be very difficult.

Trouble with Some C++ Compilers

When a C++ module needs a pointer to an object of a class, the C++ compiler does not need to know the class definition.[9] However, if it turns out later that the class actually has been instantiated from a template, Digital C++ Compiler 6.0 starts choking. It would be much easier if all commonly used objects belonged to normal classes.

The Solution: Wrapping Template Instantiations

All Standard Template Library related problems described earlier can be solved by wrapping normal C++ classes around the STL containers, which are used for storing raw pointers. For instance, a list of EXPRESSION is defined in the class EXPRESSIONLIST, which has a private member of the type `list<Expression*>`. The destructor of EXPRESSIONLIST will take care of deallocating the list member objects.

Wrapping STL containers in this way also has other advantages. The programmer has full control over the methods defined for the wrapper class. Not all methods of the container need to be made available, and they can be renamed. It is also possible to define additional methods, e.g. for dumping the syntax tree node (see Section 6.1).

In our SDL compiler, STL template instantiations were originally encapsulated by deriving other classes from them. It seemed to work, but there was a potential risk involving destructors. Since the destructors of STL containers are not virtual, it is not guaranteed that both the destructor of the STL container and the destructor of the class derived from it are called in all cases. Leaving such a flaky solution in the program was out of question, since it would most probably have caused memory leaks or unexpected problems later.

We tried to solve the problem by using pointer wrapper classes in some places, but there were two problems involved with them. First, a pointer wrapper without reference counting causes very much copying, most of which is unnecessary. Second, we wanted to have some custom methods in the container objects, and adding them to the STL templates was clearly out of question.

Now all STL containers in our compiler are kept in private members of wrapper classes. This also makes it possible to restrict access to the methods in the STL containers.

5.3 Semantic Analysis

A programming language definition specifies two things: syntax (what the language looks like) and semantics (what different constructs in the language mean).

[9] Remember, pointers always require the same amount of memory, no matter what kind of an object they are pointing to.

With some deliberate exaggeration, we simply say that a program is *syntactically correct* if it is accepted by the context-free grammar embedded in the particular compiler's front-end (that is, it contains no such errors that are syntactic in the sense of Chapter 4). Since programming languages are generally not context-free, syntactic correctness is not sufficient for *semantic consistence*. For example, if an operation in the program text has operands whose types are mutually incompatible, then the program as a whole has no well-defined semantics; and it is usually not reasonable to try to incorporate the required non-trivial check into a context-free grammar. Ensuring semantic consistence, or *semantic analysis* (that is, the detection of such errors that are semantic in the sense of Chapter 4), typically involves a lot of programming work, since the checks have to be hand-coded for each construct in the syntax tree.

5.3.1 Simple Analysis while Parsing

Some of the semantic analysis can take place in the parsing stage. For instance, consider the SDL construct ***SYSTEM*** ⟨*name*⟩/***ENDSYSTEM*** [⟨*name*⟩]. If a ⟨*name*⟩ is specified after ***ENDSYSTEM***, it must match the ⟨*name*⟩ following the ***SYSTEM***. Our SDL parser performs the check while reading in the SDL specification, and the second ⟨*name*⟩ is never stored in the syntax tree.

5.3.2 Resolving References

When a construct in the input is allowed to contain forward references, e.g. it is allowed to refer to data types which are defined later in the program, no semantic analysis of the construct can be completed until the whole input has been parsed.

Forward references are allowed practically everywhere in SDL. Our compiler resolves no names or identifiers until it has parsed the whole input. Syntax tree nodes will have to house the name or identifier of a referenced syntax tree node until it has been resolved. The whole syntax tree can be resolved recursively. Special measures must be taken to avoid infinite recursion. For instance, the cyclic declaration `syntype a=b endsyntype; syntype b=a endsyntype;` must be detected by the resolution algorithm.

The nested scoping and separate namespaces in SDL make the resolution of references interesting. An ⟨*identifier*⟩ consists of a ⟨*name*⟩ preceded by an optional ⟨*qualifier*⟩, which identifies the entity where the object with the name is defined. The qualifier is a list of path items, listing nested entities starting from the top level. Leftmost parts of the qualifier can be omitted, if they are common with the entity surrounding the identifier. Because of this, the identifier resolution algorithm must be prepared to deal with ambiguities. A qualifier could refer to two objects, for instance to one defined in an entity of the current package, and to another defined in an equally-named entity on the specification level.

5.3.3 Type Checking and Expression Evaluation

Evaluating SDL expressions is not straightforward. Literals are almost indistinguishable from each other. Only character string constants are distinguished from other literals in the grammar. Neither the literals nor the names of built-in sorts are reserved words. A literal `42` can belong to a user-defined (possibly enumerated) sort as well as to the built-in sorts `Integer` or `Real` or their derivatives. It can also be a variable name. In addition, built-in operators can be defined for user-defined sorts. An expression like `1+2` can be e.g. of the `Integer` sort and evaluate to `3`. There could also be a user-defined sort with literals `1`, `2` and the operator `+`, which can be defined to return anything for `1+2`.

These problems can be addressed by evaluating expressions in two stages. First, the sort of the expression is determined from the context. For instance, if an expression is assigned to a variable, the expression must have the same sort as the variable. The sort will be recursively propagated to all subexpressions of the expression, including the literals. Not necessarily all subexpressions have the same sort; e.g. the parameters of a procedure call must agree with the sorts defined in the interface of the procedure. If the sort propagation succeeds,[10] the expression can be evaluated.

These things have not yet been implemented in the SDL compiler, since it is not clear which data types will be directly supported by the reachability analyzer. Expression evaluation is tightly coupled with code generation, or Petri Net model generation in our case. Also the ASN.1 parser must be integrated with the compiler, so that data types defined in terms of ASN.1 can be used in SDL expressions.

5.3.4 Checking Signals and Connections

In SDL, processes communicate by sending messages (called signals in SDL) to each other's message queues. Each process has a set of valid input signals. If a process receives an invalid signal, the signal will be silently discarded from the message queue. Sending an invalid signal to a process can be viewed as an error. Similarly, it is an error to send a signal that is not allowed on a connection or signal route.

This kind of errors can often be discovered by the compiler. Some data flow analysis can be used to determine which process or channel is receiving the signal. If it cannot be determined at compile time, the model generator must make sure that such errors will be detected at "run time," during the reachability analysis of the model generated from the SDL specification.

[10] For instance, there could be references to undefined procedures, or a procedure could have a different number of arguments than specified. Also, a sort could lack an operator or a literal that is used in the expression.

Chapter 6

Debugging Methods

One cannot expect a non-trivial program to be free of bugs. Compilers are very complicated programs manipulating complex data structures. A subtle programming mistake can corrupt a data structure entirely. Debugging and testing are essential in all stages of development.

6.1 Dumping the Syntax Tree

The syntax tree is an utterly complex data structure. Usually compilers contain routines for canonically printing out the syntax tree of the input program. Our SDL compiler is no exception. Routines for dumping the syntax tree can be included to the program as a compile-time option.

Object-oriented programming makes dumping the syntax tree easy. Almost every class has a method `prettyPrint()` that prints out the object and its component objects. The method has one parameter, the indentation level. Figure 6.2 illustrates a syntax tree dump of the simple program presented in Figure 6.1. The type and name of each object is displayed, and the values in parentheses represent the memory addresses where the objects are located.

Displaying the memory addresses in the syntax tree dump make the output dependent on the C++ compiler and run-time libraries used for the SDL compiler, but they can ease debugging, when one calls the `prettyPrint()` methods of selected objects from the debugger.

Dumping the syntax tree is an easy way for detecting coarse errors in the pointer arithmetics of the compiler.[1] It is also a very useful aid when fine-tuning

[1] When something is terribly wrong with the syntax tree, the syntax tree dump procedure will

```
process click_here_to;
 start;
 stop;
endprocess click_here_to;
```

Figure 6.1: A simple SDL program

```
Specification (0x815c628) {
  ProcessMap {
    click_here_to Process (0x8160d48) {
    automaton:
      Automaton (0x8161398) {
      start:
        Stop (0x8160f20)
      }
    }
  }
}
```

Figure 6.2: A syntax tree dump of the program in Figure 6.1

error handling, or when trying to find out why the compiler works in a certain way. Together with the debugging mode of the Bison-generated parser, the syntax tree dump helps one locate errors in the grammar or in the semantic actions.

6.2 Regression Testing

Even though our SDL compiler does not generate anything except syntax tree dumps yet, we say a few words about debugging the generated code, which is a Petri Net model description in our case. It may be useful to have two kinds of printouts of the generated code: one for debugging and the other containing the actual output one expects to obtain from the compiler.

The code generator is one of the most critical parts of a compiler. If it produces incorrect code, the compiler is useless. When modifying the code generator, one wants to be sure that the modification has no unexpected side effects.

Large compilers have so many constructs that it is unfeasible to manually verify that all of them are translated properly. *Regression testing* is a much better solution. A set of test cases, consisting of input programs and their expected translations, is written, and each time the compiler is modified, the test cases are run. If the compiler produces unexpected output, the programmer will compare an archived copy of the expected output to the actual output of the compiler and determine whether the compiler is in error or the expected output needs to be updated.[2]

Regression testing can be implemented in many ways. It seems reasonable to keep the test suite stored in a version control system. When the code generation is modified, archived copies of the expected output will be updated to the version control system. Test suites will be synchronized with compiler releases by tagging

probably be aborted because of a segmentation violation.

[2]The test case needs to be updated when the correction of a previously undetected error in the compiler affects the output, or when optimizations are improved. In both cases, the generated code must be carefully examined to ensure that it is correct. If it is, the output of later compiler versions will be compared against it.

all files with a common symbolic name. In this way, it is possible to repeat the test suites of older compiler releases.

6.3 Assuring Portability

C and C++ are designed to be *portable* languages, meaning that in the ideal case, a program developed for one system using one standard-conforming compiler can be compiled for (ported to) any system using some other compiler.

Compilers tend to have non-standard or system-dependent features, and some compilers do not implement all of the standard. C++ and the Standard Template Library have been developed a lot in the past years, and not all compilers support all the essential features of the standard [5] yet.

To maximize the portability of a C++ program, one should try compiling it with as many compilers as possible. Some compilers are out of question, because they do not support a recent enough version of the Standard Template Library, but others require only a small amount of changes to the code.[3]

The SDL compiler has been successfully compiled using the following C++ compilers:

- EGCS 2.91.57
- Digital C++ Compiler 6.0
- Hewlett–Packard aC++ Compiler A.01.06

6.4 Detecting Memory Leaks

Dynamic memory allocation without *garbage collection*, automatic deallocation of unused objects, involves a problem called *memory leaks*. A program that dynamically allocates an object and does not deallocate it when the object is no longer needed is said to have a memory leak. The memory area occupied by the object is totally useless, since the object is not needed any more. As a result, the program will probably require more memory in run-time than an equivalent program having no memory leaks.

We debugged the memory management of our SDL compiler using several tools. One very useful tool is the Electric Fence library [19], a `malloc()` debugger library that makes use of the memory management hardware. It protects all deallocated memory against read and write accesses. Any access to unallocated memory is caught immediately.[4] Electric Fence does not detect memory leaks, though.

[3] Hewlett–Packard's C++ compiler (aC++ A.01.06) falls in the latter category: its template deduction algorithms fail sometimes, and `operator!=` does not work with `const_iterator`s. Luckily `operator==` does work.

[4] Normally such errors are not detected until much later, when some other part of the program crashes because its data structures were overwritten or the internal data structures used for dynamic memory management are corrupted.

There are not many freely available memory debugging tools. Checker [11] only works with C programs, not with generic C++. The dmalloc library [24] seems promising, but the version we tested does not produce very useful debugging messages. For each leaked memory area, dmalloc reports the line number and the file name of the `new` operator application or `malloc()` call that allocated the memory area. When using the Standard Template Library, the information is of little use, since most objects will be allocated somewhere in an STL template. A stack trace dump, listing all procedure invocations that were in effect when the memory area was allocated, would be more useful. However, generating such a dump is nontrivial and highly system-dependent.

The only commercially available memory debugging tool we tried is Third Degree [7]. When a program is run under its control,[5] a log file will be generated, reporting all memory leaks using helpful stack traces. An interactive memory debugger that would let one to analyze memory leaks while the program is running would be more convenient to use, but no such tool was available for us. Currently our parser leaks memory from several places.

Luckily, memory leaks are not very crucial in typical compiler front-ends, which merely parse the program, allocating nodes for the syntax tree. The whole syntax tree will be kept in memory until the compiler finishes, hopefully deallocating all memory areas it has allocated. When the compiler terminates, the operating system will free all memory allocated by it anyway. If the compiler allocated and deallocated memory repeatedly in a loop, the situation would be completely different, and the memory leaks would become significant. Memory leaks are far more significant in the code generation phase, especially when performing optimizations, since the generated code is likely to be reorganized or rewritten several times.

[5] The program will run hundreds or thousands of times slower than normally.

Chapter 7

Conclusions

7.1 Contribution of the Thesis

We have created a compiler front-end for the CCITT Specification and Description Language [27], so that the MARIA reachability analyzer eventually can verify the correctness of distributed systems specified in terms of SDL. Until very recently (see e.g. [12]), formal verification of a design has not been possible without manual translation of the system to a model. Compilers will make this laborious step unnecessary, and formal verification tools will become more accessible to system designers.

To the best of our knowledge, our compiler is the first freely distributable SDL compiler that supports all of the SDL/PR grammar,[1] as defined in [27]. It was able to find a previously undetected error in [27, Annex D], as described in Section 4.2.2.

Transforming the SDL grammar from [27] to a format suitable for the parser generator tool Bison [6] and implementing part of the parsing in the lexical analyzer, as described in Chapters 2 and 3, were the most challenging and time-consuming tasks in the development of the compiler front-end.

Considerable amount of work was invested in fine-tuning the parser, improving its tolerance for erroneous input and making error messages more helpful for the user, as discussed in Chapter 4. We find that a compiler producing meaningful error messages is comparable to an interactive program having a polished user interface.

The syntax tree of the compiler makes use of the Standard Template Library defined in the C++ standard [5]. This reduces the amount of code needed and improves the reusability of the code. There were some problems with the Standard Template Library. Both problems, storing objects derived from an abstract base class, and portability problems with some compilers, have been solved, as described in Chapter 5.

All in all, we are satisfied with the overall design of the compiler and believe that it has great potential, once its model generator part has been finished. As a

[1] Currently the compiler does not support macros, but it can easily be extended with a separate preprocessor later if desired.

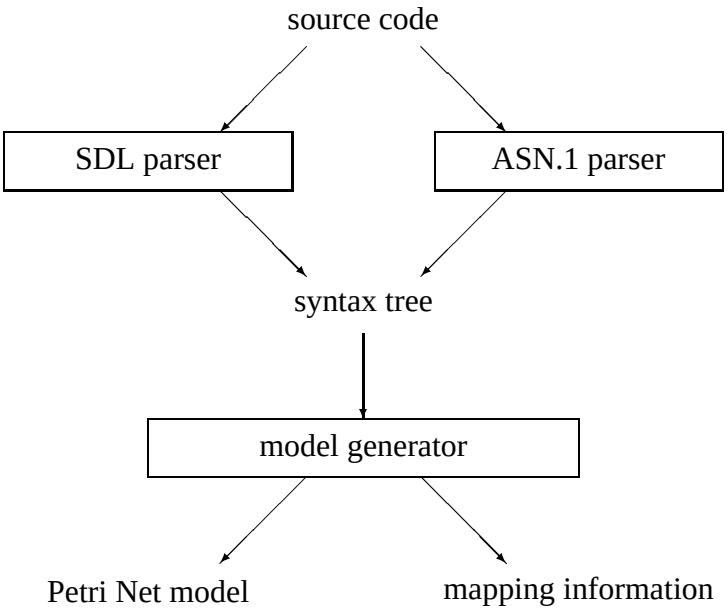

Figure 7.1: Block diagram of the SDL front-end for MARIA

freely distributable program[2] it will be used (and tested) by a heterogenous group of users around the world, and people will become more aware of the possibilities of formal verification.

7.2 Future Work

A complete SDL compiler including a model generator back-end[3] would have been too big a project to be presented in a Master's thesis.[4] What has been done so far is approximately half of a complete, non-optimizing compiler.

The next step in the MARIA project is to develop the core of the reachability analyzer. A quick and dirty version will suffice at first—different reduction methods and optimizations can be added in later. The SDL system will have two interfaces to the analyzer. The SDL front-end (Figure 7.1) will produce a model (see Section 7.2.1) that will be analyzed by the analyzer. The SDL back-end will translate the results of the analysis to a form understandable by SDL users (see Section 7.2.2). Both interfaces will use text files—the analyzer, the SDL front-end and the SDL back-end are separate programs.

[2]The software developed in the MARIA project will be available under the conditions of the GNU General Public License [10], which basically allows everyone to use and distribute the licensed software and to publish modified versions of it, provided that the modifications are also distributed in the form of source code.

[3]A Petri Net model generator back-end for an SDL-like language [15, 17] was the subject of two Master's theses.

[4]The parser and the rudimentary semantic analysis routines written so far consist of more than 20,000 lines or 500 kilobytes of source code.

7.2.1 Model Generation

Generating a high-level Petri Net model of an SDL specification involves much work, but it is rather straightforward and has been done previously for TNSDL e.g. in [13, 15, 17]. Figure 7.1 represents the outline of the model generator front-end. The SDL specification, optionally making use of data types defined in ASN.1, will be converted to a syntax tree representation by the SDL and ASN.1 parsers. The model generator produces two outputs: a high-level Petri Net model corresponding to the SDL specification and some reverse-mapping information, which is required by the SDL back-end (Section 7.2.2).

The model generator does not need to support all aspects of SDL at once—a limited set of constructs will make it possible to test the model generator (and the reachability analyzer) with simple SDL specifications.

It is useful to have one additional level of abstraction in the model generation. When the Petri Net model is first generated in memory, using place and transition objects, the compiler can support different output languages. There are several tools for analyzing high-level Petri Nets, and each tool has its own input language. Output methods will be implemented for each supported net description language. This makes it possible to compare the analyzer being developed in the MARIA project with competing approaches.

Mapping Data Types

Modern programming languages support very sophisticated data types, while tools for reachability analysis often are limited to tuples of integers or to types that can be easily represented with integers having a limited range. In [17], TNSDL data types were mapped to tuples of integers.

Union types, types whose values belong to one of the component types defined for the union type, are commonly used in protocol specifications. Standard SDL [27] does not support union types, but the combination of SDL and ASN.1 [28] does. The data type mapping used in [17] is not suitable for union types.

The new reachability analyzer of the MARIA project must support the data types presented in Tables 7.1 and 7.2, or there must exist an efficient mapping between these types and the internal data types of the analyzer. The analyzer should support ASN.1-like constraints for all data types. The Real type is not suitable for analysis, unless its precision and range are limited.

Also the Array type is going to be difficult for the analyzer, because the indexing type can be anything. It can be implemented with a dynamically growing hash table à la Perl [23], or with pairs of index types and item types. The latter solution is utterly inefficient, and it does not prevent duplicated array entries. Array elements can be initialized in an effective way by assigning a *default expression* to the array. Only elements that differ from the default value (which may be calculated from the index value) will be physically stored in the reachability graph.

Uninitialized variables are handled with a special value “uninitialized.” Any attempt to read an uninitialized variable (or Array element or Struct component) will be reported as an error in the reachability analysis.

Table 7.1: Simple data types required in the analysis of high-level languages

Integer	At most 32-bit or 64-bit signed or unsigned integer
Boolean	Boolean truth value
Enumerated	Type with named literals
Character	A single 8-bit character
Real	Floating point number
Id	Type for retrieving unique object identifiers

Table 7.2: Complex data types required in the analysis of high-level languages

String	Variable-length array of any type indexed by integers
Array	An array of any type indexed by values of any type
Struct	Structured data type with named and typed components
Union	Collection of alternative types

Values of the Union type will be tagged, so that the actual type of the Union value is known at all times. Implicit type conversions are not supported: writing an Integer value to a Union type and attempting to assign it to a Boolean variable is an error. Union could also be implemented as a Struct with all components except one being uninitialized. In this case, special measures should be taken to ensure that only one component is initialized at a time. Similar to Array elements, each Struct component will be assigned a default value, which must be a literal of the type of the component or the special value “uninitialized.” Also String items and simple data types can have default values. Default values for simple data types will only apply when the type is used by itself, i.e. not as a component of a complex data type.

Template Instantiation

SDL contains some high-level constructs that have to be expanded to equivalent lower-level constructs before any compilation can take place. Support for the high-level constructs, such as system, block, process and service types, has low priority when the code generator is developed. First there must exist a translation from the underlying lower-level constructs to the target language. Then, after the translation for the higher-level construct has been implemented, it is fairly easy to compare the target code generated for a program using the higher-level construct with the translation of another program using an equivalent lower-level construct.

7.2.2 Representing the Analysis Results

When the reachability analysis finds an error in the model generated from the SDL specification, its error report, consisting of cryptic, automatically generated place

and transition names, is Pig Latin for the engineer who wants to know what is wrong with his design. The error report from the analyzer effectively only tells him: “There is an error somewhere in the design.”

For each state that the engineer is interested in,[5] values of all system variables need to be extracted, and each transition from one state to another should be associated with a statement in the SDL specification. This task either requires some additional reverse-mapping information from the model generator, or the places and transitions in the model must be named so that the SDL variable names and execution points can be deduced from them. [17] presents one solution to this problem.

[5] Typically one wants to examine the error state and a few states preceding it, or the states on the shortest path from the initial state to the error.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison–Wesley, Reading, MA, USA, March 1986. 4, 7, 11, 19, 31, 37

[2] Brian Berliner. CVS II: Parallelizing Software Development. In *Winter 1990 USENIX Conference*, pages 341–352, Washington, DC, USA, 1990. 5

[3] Jonathan Billington. Many-Sorted High Level Nets. In *3rd Workshop on Petri Nets and Performance Models*, pages 166–179, Washington, DC, USA, 11–13 December 1989. IEEE CS Press. 2

[4] *Programming Languages—C*. ISO/IEC 9899. International Organization for Standardization, Geneva, Switzerland, 1990. 2, 5

[5] *Programming Languages—C++*. ISO/IEC 14882. International Organization for Standardization, Geneva, Switzerland, 1998. 5, 5, 59, 61

[6] Robert Corbett, Richard Stallman, and Wilfred Hansen. *The Bison Reference Manual*. Free Software Foundation, 1995. Version 1.25. 4, 10, 61

[7] Jeremy Dion and Louis Monier. *Third Degree: heap usage and leak profiler, and memory-access error checker for C and C++ programs*. Digital Equipment Corporation, 1998. 60

[8] Chris Dodd, Vadim Maslov, et al. BTYACC—*Backtracking Yacc*, 1997. Release notes for Version 2.1; ⟨URL:http://www.siber.com/btyacc/⟩. 4

[9] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL—Formal Object-oriented Language for Communicating Systems*. Prentice Hall, London, England, 1997. 3, 38

[10] Free Software Foundation. GNU General Public License, June 1991. Version 2. 62

[11] Tristan Gingold. *Checker, a memory access detector*, 1996. Documentation for Version 0.8. 60

[12] Bernd Grahlmann. The PEP Tool. In O. Grumberg, editor, *Proceedings of CAV'97 (Computer Aided Verification)*, volume 1217 of *Lecture Notes in Computer Science*, pages 65–80, Haifa, Israel, June 1997. Springer-Verlag. 61

[13] Nisse Husberg. Verifying SDL Programs using Petri Nets. In *SMC'98 Conference Proceedings*, volume 1, pages 208–213, San Diego, CA, USA, October 1998. Institute of Electrical and Electronics Engineers, Inc. 63

[14] Cygnus Incorporated. EGCS Frequently Asked Questions, September 1998. ⟨URL:http://egcs.cygnus.com/faq.html⟩. 5

[15] Tero Jyrinki. Dynamic Analysis of SDL Programs with Predicate/Transition Nets. Technical Report B17, Helsinki University of Technology, Theoretical Computer Science Laboratory, Espoo, Finland, April 1997. 62, 63

[16] Andreas Lutsch. Ein Parser für SDL. Jahresarbeit, Humboldt-Universität zu Berlin, Fachbereich Informatik, Berlin, Germany, November 1993. 4, 13, 14

[17] Markus Malmqvist. Methodology of Dynamical Analysis of SDL Programs Using Predicate/Transition Nets. Technical Report B16, Helsinki University of Technology, Theoretical Computer Science Laboratory, Espoo, Finland, April 1997. 2, 62, 63, 63, 63, 65

[18] Vern Paxson, Van Jacobson, Jef Poskanzer, and Kevin Gong. *Flex—fast lexical analyzer generator*, April 1995. Version 2.5. 4, 21

[19] Bruce Perens. *Electric Fence Malloc Debugger*. Pixar Animation Studios, 1993. Manual page for Version 2.0.5. 59

[20] Wolfgang Reisig. *Petrinetze—Eine Einführung*. Springer-Verlag, Berlin, Germany, 1986. 2, 46

[21] *Standardization of basic model page-printing machine using International Alphabet No. 5*. Recommendation S.30. International Telecommunication Union, Geneva, Switzerland, 1988. 23

[22] Walter F. Tichy. RCS: A System for Version Control. *Software—Practice and Experience*, 15(7):637–654, July 1985. 5

[23] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Cambridge, MA, USA, 2nd edition, September 1996. 63

[24] Gary Watson. *Debug Malloc Library*, January 1998. Documentation for Version 3.3.1. 60

[25] Niklaus Wirth. What Can We Do About the Unnecessary Diversity of Notation for Syntatic Definitions. *Communications of the ACM*, 20(11):822–823, November 1977. 9

[26] *Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Recommendation X.680. International Telecommunication Union, Geneva, Switzerland, July 1994. 2, 4, 4

[27] *CCITT Specification and Description Language (SDL)*. Recommendation Z.100. International Telecommunication Union, Geneva, Switzerland, October 1996. 2, 4, 4, 7, 8, 8, 14, 16, 19, 19, 22, 23, 34, 34, 35, 37, 38, 44, 47, 61, 61, 61, 61, 63

[28] *SDL combined with ASN.1 (SDL/ASN.1)*. Recommendation Z.105. International Telecommunication Union, Geneva, Switzerland, March 1995. 2, 3, 63

[29] *Common interchange format for SDL*. Recommendation Z.106. International Telecommunication Union, Geneva, Switzerland, October 1996. 2