Automatic GUI creation

– Generating Java source code from formal descriptions

Master’s thesis in Computer Science

Per Cederberg, d93-pol@nada.kth.se

21st of March 2000

Supervisor: Viggo Kann (viggo@nada.kth.se)
Department of Numerical Analysis
and Computer Science
Royal Institute of Technology

Supervisors at Ericsson Telecom:
Kaj Bjurman (qtxkaj@etx.ericsson.se)
Arvid Svensson (qtxarvs@etx.ericson.se)

ABSTRACT

This thesis evaluates the possibility to automatically create a graphical user interface (GUI) from a formal description of the fields it should contain. A prototype has been developed for a limited domain of applications—interfaces of configuration and maintenance components. The prototype reads a formal description in the ASN.1 syntax of the fields in the interface, allows some interactive changes to the interpretation, and outputs the Java source code that creates the interface.

The generated source is to be used when developing plug-ins for an operation and maintenance framework, and special care has been taken to make the code as human readable and understandable as possible. This effectively means following strict quality norms—indenting, commenting and modularizing the source code, as a human programmer ought to do.

The results obtained from the prototype have been very encouraging, indicating that large parts of the interface can actually be generated automatically. The prototype also generates some other parts of the plug-in, which helps structuring the rest of the code. The usage of an automated tool generally reduces the amount of tiresome and repetitive work, while also adding more robust input verification and avoiding some typical bugs.

Automatisk generering av grafiska gränssnitt

– från formella beskrivningar till källkod i Java

SAMMANFATTNING

Denna rapport utvärderar möjligheterna att automatiskt skapa ett grafiskt användargränssnitt (GUI) från en formell beskrivning av de ingående fälten. En prototyp har utvecklats för en begränsad mängd applikationer – gränssnitt till konfigurerings- och underhållskomponenter. Prototypen läser in en formell beskrivning i ASN.1 av gränssnittets fält, tillåter några interaktiva ändringar i tolkningen och skriver ut den Java-källkod som skapar gränssnittet.

Den genererade koden ska användas när man utvecklar komponenter till ett konfigurerings- och underhållssystem. Speciella hänsyn har tagits för att göra koden så läsbar och förståelig som möjligt för ett mänskligt öga. Detta har inneburit att strikta normer för kvalitet har följts – koden har indenterats, kommenterats och modulariserats på samma sätt som en mänsklig programmerare borde göra.

De resultat som uppnåtts med prototypen har varit uppmuntrande, indikerande att stora delar av gränssnittet faktiskt kan genereras automatiskt. Prototypen genererar också några andra delar av komponenten, vilket hjälper till att strukturera den återstående koden. Att använda ett automatiskt verktyg innebär generellt att mängden tröttsamt och repetitivt arbete kan minskas, samtidigt som mer robust inmatningsverifiering kan läggas till och några typiska fel undvikas.

TABLE OF CONTENTS

1. Introduction

1.1 Background

The Ericsson IP telephony system consists of several distinct components, each having its own control and configuration demands. As an important part of the system, the operation and maintenance (O&M) subsystem consequently must be able to handle this mixed set of network elements and configuration parameters.

The upcoming O&M system will have a strongly modularized architecture, delegating the actual management functionality to specialized plug-ins. Thus, in order to handle a new type of network element, a plug-in must be developed and installed into the O&M framework. Each plug-in, in turn, must be divided into several layers of functionality. These layers are typically responsible for graphical user interface (GUI), interface logic, data backup storage, etc.

1.2 Problem

The development of plug-ins is cumbersome, and major parts of the development are suitable for automation. By using information from the formal descriptions for each element, a functioning skeleton plug-in can be created automatically, reducing development time by an order of magnitude. Apart from speeding up the development process, automatic code generation also has other benefits—similar structure will be imposed on all plug-ins (having significant maintenance benefits), interfaces will use standardized components, and some types of bugs will be avoided to a large extent.

Automatically generating source code is not trivial, however. The GUI component(s) can be guessed from the formal descriptions, but the selections made this way must be easy to change. Also, some data representations in the formal descriptions are not well chosen with respect to their actual contents and the mapping between data representation and GUI component is neither constant nor one-to-one.

Clever methods must be used for approximating the component choice, allowing the component to be changed dynamically, and converting underlying data to the format suitable for the chosen component. Ideally the automated process should also allow iterations, as already generated plug-ins may have to be regenerated when specifications change. The goal must be that automatically generated code should be at least as easy to maintain and extend as source code written entirely by hand.

There are several limitations to the results that can be obtained by automatically generating plug-ins. User interface layout and program logic are two examples of parts that can hardly be created automatically. Neither are highly specialized plug-ins a target for automation. The generated code can only be a support for the plug-in programmer, as several parts of the plug-in cannot be created well by automatic means.

1.3 Purpose and method

The main goal with this thesis was to show how far the automatic generation of the plug-in GUI could be pushed, and to implement these parts in a prototype application. Additionally, other parts of the plug-in creation have also been explained briefly and/or implemented in the prototype.

This thesis will primarily focus on three parts—analysis of the formal element descriptions, selection of components and other data structures, and code generation. Emphasis has been put on theory and design decisions, not on implementation details. For further information on the actual implementation, please refer to the source code comments.

The prototype application has been implemented in Java, and contains a set of solutions to the main problems. The prototype generates general, syntactically correct, indented and commented Java source code with some changes being necessary to fully fit the plug-in framework.

2. Analyzing Element Descriptions

In this section the structure and analysis of the formal element descriptions is explained. The section consists of four parts—the three first containing background material. The first part introduces the element descriptions and their syntax. The second part explains how grammars work and how they can be used to analyze a character stream. The third part deals with implementing such an analysis in Java. The last part, finally, reveals the techniques used and design decisions made in the prototype analyzer.

2.1 Formal element descriptions

Most of the elements in the Ericsson IP telephony system are configured and managed remotely with the *Simple Network Management Protocol* (SNMP, Case et al., 1990). SNMP is most commonly used to configure routers, gateways and other network elements on the Internet, and is an Internet Activities Board recommendation described in RFC 1157.

The SNMP standard requires all manageable elements to have their interface described formally in a *Manageable Information Base* (MIB), a format described in RFC 1155 (Rose & McCloghrie, 1990). Each MIB typically contains a list of field and table names associated with their data types, access conditions and unique identifiers.

The ASN.1 syntax

The MIB files are normal text files written in *Abstract Syntax Notation One* (ASN.1, ISO 8824), an ISO standard for declaring variables, composite types and other structures. Actually only a subset of the full ASN.1 syntax is used in the MIB files, as defined in RFC 1155 (Rose & McCloghrie, 1990). The fields must also be declared using the macros defined in RFC 1212 (Rose & McCloghrie, 1991) and RFC 1215 (Rose, 1991).

The ASN.1 language is not a full programming language, but only contains type and structure declarations. Specialized encoders then use the structural declarations when encoding or decoding the actual data, the idea being that declarations and low-level representations should be separated. Currently there exists a number of such encoders, the most commonly undoubtedly being the Basic Encoding Rules (BER) used with SNMP.

Two simple ASN.1 examples

The syntax of ASN.1 is somewhat different from that of a normal programming language, as can be seen in example 1. The example shows a simple declaration, written with the RFC 1212 extensions to the pure syntax. It is worth noting the lack of delimiter characters, something that makes recovery from input errors more difficult.

```
cfgNumberLength OBJECT-TYPE
        SYNTAX INTEGER
        ACCESS read-write
        STATUS mandatory
        DESCRIPTION
        "The number of digits that must follow the VG route
        number in order to form a full number."
        DEFVAL { 3 }
::= { config 1 }
```

Example 1. Declaration of an integer field `cfgNumberLength` with a default value of 3. The code also contains the access rights, the implementation status, and a short description. Finally a unique identifier is defined, in order to avoid referring to the field by its name.

A more complicated declaration is shown in example 2, at least from the perspective of automatic code generation. The field is declared as having a data representation that does not correspond to the actual data content, causing the automatic code generator to select an inadequate or ill suited visual presentation. Overcoming such difficulties is hardly possible without the usage of sophisticated heuristics or full natural language processing.

Some disadvantages with ASN.1

ASN.1 has several disadvantages compared to structure declarations in common programming languages, such as C or Pascal. The declarations tend to be large, non-compact and cluttered with keywords. Much repetition is also required for composite structures, as each part is declared twice—first with name and type inside the composite structure, and then afterwards with all the required parameters.

The typical ASN.1 file is rather large, which stems in part from the use of long keywords like `OBJECT-TYPE`, `SYNTAX` and `DESCRIPTION`—being easy to read for the untrained eye, but requiring much unnecessary and error-prone copying when writing the MIB files.

```
cfgLGK1 OBJECT-TYPE
        SYNTAX DisplayString
        ACCESS read-write
        STATUS mandatory
        DESCRIPTION
        "This is the IP address of the primary Local
        Gatekeeper used by this voicegateway.
        The address takes the form X.X.X.X where
        X is an integer in the range of 0 and 255."
        DEFVAL { "" }
::= { config 2 }
```

Example 2. Declaration of a normal string field `cfgLGK1` containing characters. The string contents should be interpreted as forming a valid IP address, something only mentioned in the description. This is a typical case when the automatic code generation will not be able to present the field data in an adequate way.

More implementation relevance has the free ordering of the constructs in ASN.1, something that cause much trouble when analyzing MIB files. The analysis must normally be made in at least two passes, as identifiers may be read before being defined or declared.

Another serious implementation problem is the fact that the basic types have no size limitation to their contents. Ranges can optionally be specified, but if no limit is specified the ranges are always supposed to be infinite. The use of infinite integers, infinite precision floats, and infinite length strings inevitably cause implementation difficulties, as all actual programming languages specify limits to their basic types. If special care is not taken in all steps of implementation, the result may well be hidden size dependencies that can be hard to track down.

2.2 Grammars and parsing

In order to analyze a stream of characters a special technique, called *parsing*, is normally used. The parsing is divided into three steps, or passes—lexical analysis, syntactic analysis, and semantic analysis. In the lexical analysis, adjacent characters are grouped together into *tokens*, more or less like letters are grouped together into words when we read a normal text.[1] In the syntactic analysis the stream of tokens is analyzed according to a grammar and a *parse tree*, i.e. a hierarchical tree grouping the tokens, is created. In the semantic analysis, finally, the information is extracted from the parse tree and converted into some internal representation. During all three stages errors in the input can be detected and reported.

[1] This is intended to reflect an intuitive idea on how we read texts, and is not to be understood as a proven fact. Psychological studies, on the contrary, show that the reading process is mostly based on direct image recognition of the words. This is, however, outside the scope of this thesis.

Grammars and productions

The grammar is absolutely central to the parsing as it controls the input syntax and the structure of the parse tree (sometimes also called *syntax tree*). The choice of grammar also controls the parser behavior, as different grammar formats implies different implementation techniques. But to understand this we must first learn more about grammars.

A grammar is said to consist of a series of *productions*, i.e. rules, controlling the structure of the input. As an example a sentence production could state that a sentence should consist of a noun-phrase, a verb-phrase and a period (see example 3). These parts in turn can be either tokens or references to other productions in the same grammar. The complete grammar consists of all the productions and tokens used therein.

```
Sentence ::= NounPhrase VerbPhrase "."
```

Example 3. A BNF description of a sentence consisting of a noun-phrase, a verb phrase, and a period. “Sentence” is the production being defined. “NounPhrase” and “VerbPhrase” are names of other productions (not shown here), and “.” is a reference to a period token.

The grammar itself is normally expressed in BNF (*Backus-Naur Form*), which is a syntax for specifying context free grammars. Each production in BNF consists of a two parts—the production name and it’s definition—separated by a “`::=`” symbol. The definition of the production (to the right of this symbol) consists of the sequence of tokens and productions that defines it.

The productions can also be more complicated, as would be the case of a paragraph that consists of one or more sentences. In this case two alternative productions would be needed to express the full relation, as shown in example 4. The productions may also refer to themselves in various ways.

```
Paragraph ::= Paragraph Sentence
            | Sentence
```

Example 4. Two productions expressing that a paragraph must consist of one or more sentences. Note the usage of “|” to simplify writing alternative productions with the same name.

Different types of grammars

There are various ways to express the grammar, even if using only strict BNF. One of the major differences is if the productions in the grammar are left- or right-recursive, i.e. if the productions may refer to themselves in the first or in the last position. Other differences include the number of tokens that have to be known in advance to select among the alternative productions, and whether the input string should be read from left-to-right or from right-to-left.

The two main families of grammars are the *LL* and *LR* grammars; both being read from left-to-right (which is what the first L means). The LL grammars are right-recursive, and in contrast the LR grammars are left-recursive. This is shown more clearly in example 5.

```
Paragraph ::= Paragraph Sentence
            | Sentence

Paragraph ::= Sentence Paragraph
            | Sentence
```

Example 5. Two different possibilities to express the paragraph rule from example 4. The first version is a LR form of the rule, while the second is the corresponding LL form.

The number of “look-ahead” tokens needed to choose among the alternative productions with the same name is normally stated in a parenthesis after the grammar family, as in LL(1), for a LL grammar with one token look-ahead. When specifying this number, the whole grammar must be analyzed and the highest number of tokens needed is the number that should be stated.

The LR grammars may still contain parts causing conflicts between alternative productions, at least for the type of parsers in use today. Therefore a subset of LR, called LALR, that contains some small limitations is normally used instead. As implementation complexity also rises with the usage of more look-ahead tokens, the two types of grammars most used in practice are LL(1) and LALR(1).

More ASN.1 disadvantages

The standard ASN.1 grammar is neither LL(1) nor LALR(1), normally requiring some rewriting before a parser can be implemented (Rinderknecht, 1995). The main problem is the ambiguities—the need for a look-ahead of several tokens, some identical tokens having different names, and the use of empty productions.

```
ObjIdComponent    ::= NameForm
                    | NumberForm
                    | NameAndNumberForm

NameForm          ::= <identifier>

NameAndNumberForm ::= <identifier> "(" NumberForm ")"
```

Example 6. A part of the original ASN.1 grammar showing the need for more than one token look-ahead. When inside the `ObjIdComponent` production and having an identifier as the next token, two rules may be chosen.

The usage of a look-ahead of several tokens is shown in example 6 and causes much trouble when attempting to implement a simple parser. More care has to be taken when creating a parser that handles a look-ahead of more than one token, and it also slows down parsing considerably.

Two other issues in the ASN.1 syntax that cause problems are the difficulty to recover from errors and the language extensibility. In many languages declarations are separated with the ";" character, allowing the parser to read until the next ";" upon error in the input. In ASN.1 the only separation is spaces, tabs and newlines, making error recovery difficult. The ASN.1 syntax can also be dynamically extended as macros are defined, something having the potential to make the parser infinitely complex, should one try to implement it. Most commercial ASN.1 parsers seem to contain only a set of macros, but ignoring the full macro syntax.

2.3 Writing a parser

There are several approaches to writing a parser, but the most common is undoubtedly to use a *parser generator*, i.e. a program that from a grammar can generate the source code of the parser. Two of the most popular parser generators are the lex and yacc tools (or the GNU equivalents flex and bison) that generate C source code for the lexical and syntactic analyzer. The lex and yacc tools also exist in versions that generate Java source code (JLex and Cup), but these versions offer no integration between the lexical and syntactical analyzers (Berk, 1997).

The classical tools for parser generation also have several other problems, especially when used in an object-oriented environment. For an example, they don't support automatic parse tree generation, something that would have required too much memory at the time when they were developed. Today, however, memory is not a scarce resource, and multiple pass compilers have become more common. As a result of the lack of parse tree creation, semantic analysis and action code must be inserted in the grammar, making the separation between syntactic and semantic analysis hard to impossible. This mixing up of the later stages in the parsing typically causes problems with maintainability, extensibility and complexity.

Object-oriented parsing

In a modern object oriented language a different approach is called for, especially one that uses the object oriented techniques for encapsulation of data and functions in separate objects. It is also desirable to separate the syntactic and the semantic analysis in modules, making it possible to do small syntax changes without having to change the semantic analysis, and, more important, being able to change the analysis without regenerating the entire parser.

One approach to obtain a more object-oriented parser, is by using delegating compiler objects (Bosch, 1996). The grammar is then split into smaller parts, each part being handled by a separate compiler (or parser) object. When parsing the control is delegated between the various objects, effectively obtaining an object-oriented modularization.

The approach with delegating compiler objects makes the grammar more extensible, as only small changes must be made to add a new parser object. Changes in the compiler are also made simpler and safer, as modules are smaller and easier to overview. The main drawbacks are that no special division between syntactic and semantic analysis is made, and that no such parser generator tools are available for Java.

Another way to create an object-oriented parser is by generating a full parse tree, thereby being able to separate the syntactic and semantic analysis. The object-oriented paradigm also supplies a natural and secure model for the parse tree implementation, with each node as a separate object. This allows for more secure changes in the tree structure, reducing the risk for inconsistencies and errors.

JavaCC

The most commonly used parser generator for Java is probably JavaCC (previously called Jack), made freely available for binary downloads by Sun Microsystems (Sun, 1999). It generates a recursive-descent parser in Java from a LL(n) grammar, allowing look-ahead of multiple tokens when choosing productions. The grammar is written in a special format that allows Java code to be added to the grammar productions, much in the same way as when using yacc. It is also possible to define grammar rules in the form of pure Java code, although this is not a recommended practice.

The parser generated is divided into several classes that may be replaced or extended by the user. It is also reasonably fast, and generates understandable messages upon errors in the input. However, there is only a rudimentary support for error recovery during parsing. The parser will generally fail upon the first error, making it difficult to report several errors in a single iteration.

A more fundamental problem with JavaCC, though, is the lack of support for generating parse trees. When inserting the parse tree generation code into the grammar files, the latter becomes messy and cluttered with Java source code all over. This design choice is probably an inheritance from the earlier, more primitive parser generator tools for imperative languages and older systems.

There are a couple of tools that can add code for parse tree generation to the JavaCC grammar files automatically, but none of them have proved very convincing when used with the ASN.1 grammar. In general the parse trees produced are not strongly typed, difficult to extend, and not adapted to the application at hand. The extra step when generating the parser is also irritating, and may cause some problems with reserved keywords, etc.

SableCC

A more modern approach to parser generation has been taken by SableCC, a parser generator developed by Éntienne Gagnon at the McGill University (Gagnon, 1998). SableCC is available both as binary and source code, and is released under the GNU GPL license, thereby allowing others to modify and further develop the code.

SableCC generates a table-driven bottom-up parser for a LALR(1) grammar, just as the classical lex and yacc tools. It has many advantages compared to JavaCC, most notably being the automatic generation of the parse tree. The generated tree is also strongly typed, i.e. each grammar production is mapped onto a separate class. The child nodes can thus be accessed with specific methods in each class, having the name of the child node type. The parse tree generated is thus more robust to syntax changes, as node positions in productions may be changed freely.

The current version of SableCC has a number of disadvantages, though. The grammar can only contain lowercase production and token names, the token definitions are complex, and only strict LALR(1) grammars are accepted. Furthermore there is no support for error recovery, and the error messages generated during parsing are cryptic and filled with implementation details. Finally, as a result of the creation of named access methods for all child nodes in the parse tree, all alternatives in the grammar must be tagged with additional names to become uniquely identifiable. This includes both productions having the same name, and multiple identical references within a production. The usage of such tags in the grammar files can sometimes make them difficult to read.

2.4 The prototype MIB parser

When implementing the final parser for the prototype several small tests with both JavaCC and SableCC were performed, in order to decide which one to use. JavaCC was finally chosen as it would be easier to use and produce better error messages. Using SableCC would have required too much rewriting of the grammar, causing undesired complexity where none would have been needed, and it would also have rendered the parser less usable, as the error messages would have become more cryptic and difficult to understand.

The choice of JavaCC also came with the additional cost of implementing the parse tree generation, as it was important to attempt to split the syntactic and the semantic analysis into separate modules. Apart from this the grammar had to be rewritten and a full semantic analysis with error checking and information extraction had to be implemented.

Rewriting the grammar

One of the first problems encountered when implementing the parser was the lack of a good and freely available ASN.1 grammar. The first prototype was based on a grammar extracted from the lex and yacc sources of *Snacc*, a freely available ASN.1 compiler. Later the complete ASN.1:90 grammar (in Rinderknecht, 1995) was encountered and the necessary changes to correct some errors and add missing constructs could be made.

In order to create a reasonable simple ASN.1 parser, some parts of the grammar were rewritten—removing the possibility to define macros, but adding two widely used SNMP macros. The more compact EBNF syntax was used, instead of the standard BNF, in order to save some productions. The resulting grammar (the one implemented) can be found in appendix A.1.

Syntax analysis and parse tree generation

The rewritten grammar also had to be converted to the special format used by JavaCC, and Java code to create the parse tree had to be added. Basically semantic action code was added to each production in the grammar, creating parse tree nodes as each production is parsed. Only the nodes needed in the later stages of analysis were included in the generated parse tree, thereby effectively pruning the tree already upon creation.

The solution to create the parse tree "by hand" inside the syntactic analysis became necessary when using JavaCC, but proved to generate a syntax analyzer that was hard to read and modify. The parse tree, on the other hand, became well adjusted to the task at hand and could easily be modified when needed.

The parse tree framework was strongly influenced by the models found in SableCC (Gagnon, 1998, p. 52ff). Each node in the parse tree was represented by a separate object, and support for parse tree walker (or iterator) classes was also implemented. Having separate tree walker classes made the implementation of a multi-pass semantic analyzer fairly simple and straightforward, as all the details of tree walking had been isolated in separate classes.

One major difference in the SableCC model, however, is that the parse trees use strong typing, i.e. each production and token type is implemented in a separate class. The prototype parse trees, on the other hand, do not use strong typing, as generating that number of classes by hand would have been a too repetitive and error-prone work. Instead each node is assigned a unique number representing it's type, and child nodes are accessed through general-purpose methods.

Semantic analysis and data structures

After the creation of the parse tree structure, the semantic analysis processes the tree in two passes. In the first pass all symbols (field names, types, etc) are identified and created, and in the second pass they are filled with actual information. This division was needed as symbols may be used before being declared.

The symbols created during the semantic analysis were modeled closely after the ASN.1 symbol structures, making the analyzer fairly simple and easy to implement. This design choice proved to have disadvantages later on, as it became difficult to separate different types of symbols from each other. Also some special ASN.1 deficits were inherited, as for example the table symbols having only indirect access to the column data types.

Some changes and additions were made to the original model as the prototype developed, but these were not quite enough to make symbol handling easy in the rest of the prototype. There is still need for a redesigned symbol structure, especially one where different types of symbols are split into a class hierarchy. The creation of these more specialized symbols should probably be placed as a third pass in the semantic analysis, converting all the old symbols to more specialized ones.

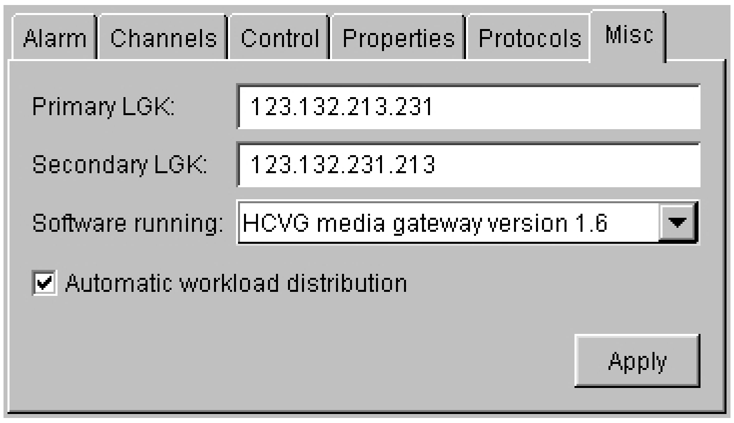
3. COMPONENT SELECTION

This section explains how the prototype application selects graphical components for the plug-in interface. The first part details how a plug-in interface is structured and how the MIB fields are mapped to components. The second part describes how the user can interact and change components while creating the plug-in. A number of screenshots from the prototype are also included.

3.1 The plug-in interface

A plug-in in the operation and maintenance system should be divided into several layers of functionality. In the lowest layer the actual SNMP transactions are handled, and in the highest layer data is presented to the user in a graphical user interface (GUI). Between these two extremes are the other layers, responsible for interface logic, temporary data storage, and data dependencies.

In a standard MIB file the number of fields is quite high, making it impossible to show them all at once. A normal plug-in interface therefore consists of a set of tabs, as can be seen in figure 1. Each tab contains a reasonable number of related fields with some usable layout.



Primary LGK: 123.132.213.231
Secondary LGK: 123.132.231.213
Software running: HCVG media gateway version 1.6
☑ Automatic workload distribution

Figure 1. A plug-in interface is normally a set of tabs, as the amount of information presented to the user would otherwise be overwhelming. The interface on each tab is usually more complex than in this simplified figure.

Java interface components

A graphical user interface in Java is constructed as a hierarchy of special objects called components. Each component can be either simple, as a label or a text field, or a compound of other components, as a tab panel. The simple components are layout inside the compound ones using special layout information that can be provided inside the program.

There are two component hierarchies in Java—the older AWT components, and the newer Swing components. Both are included in the modern Java platforms and the Swing components can also be mixed with the AWT components without much trouble. As Swing is a newer and more feature-rich solution it is the recommended choice for most applications, especially if they are not required to run from any old web browser.

The plug-in interfaces use the Swing components as they provide a richer set of features and are more extensible. Some additional components have been written to handle specialized data, as numeric IP addresses, and to fill some gaps missing in the Swing framework. Especially input verification in the components is needed for the plug-ins.

From fields to components

It is worth noting that many data fields require several components in the interface, as labels are sometimes necessary to for provide information about the data presented (see for example figure 1). This means that the mapping from MIB fields to components in the plug-in interface cannot be one-to-one, as several components will have to be generated on some occasions.

The mapping from data type to component is not always obvious, and on some occasions it may also be impossible to do by automated means, at least with a satisfying result. The choice of interface components is normally a decision that involves aspects of human-computer interaction, which means that it is not easily formalized into simple rules. The result of the automatic mapping from data type to component will therefore sometimes be mistaken, as it does not take the data semantics and context into account.

Table 1. The default mapping from field to component depends on both data type and access. In the case of integers with enumerated values, the value labels are also checked for common “boolean” substrings (“yes”, “true”, “on”, etc). The cells marked with (*) match any alternative.

Data type	Access	Component
String	read-write	JTextField
String	read	JLabel
Integer	read-write	JIntegerField
Integer	read	JLabel
Integer with enumerated values (boolean alternatives)	read-write & read	JCheckBox
Integer with enumerated values (other alternatives)	read-write	JComboBox
Array	(*)	JTable
(*)	write	JButton

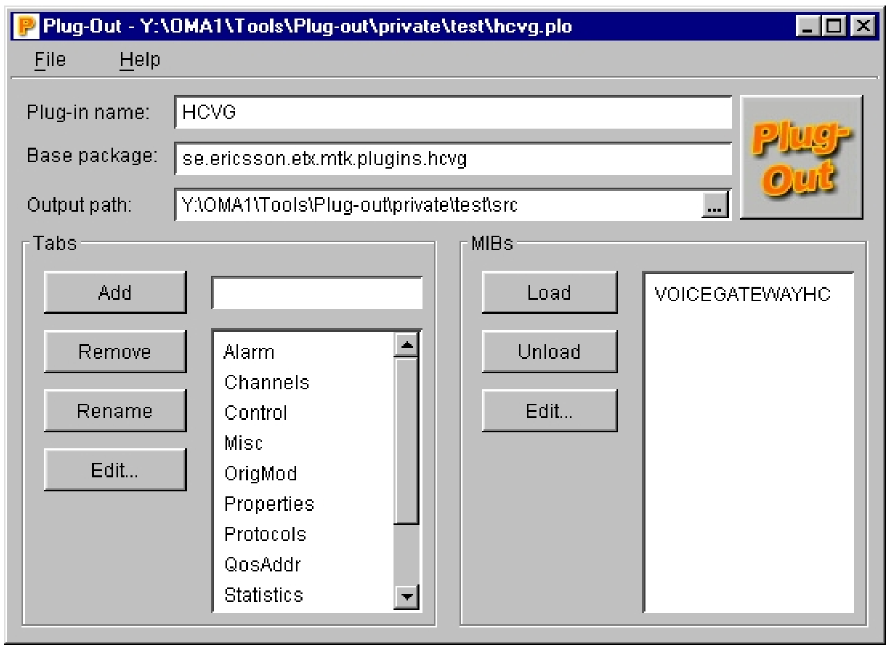
In table 1 the default mapping from data type to component is shown. Some of the components specified in reality also refer to additional labels, as such are needed in some cases. The mapping is not trivial and contains several special cases for data that may only be read.

3.2 Interaction in the prototype

Although a lot of information can be extracted from the MIB files, it is not reasonable to generate the plug-ins in a fully automated fashion. The grouping of the fields rarely corresponds to the structure in the MIB files, and the default field to component mapping may cause problems. It is therefore hardly sufficient with the simple command line syntax most compilers have, as such an interface would be far too complex.

Instead, the interaction called for is that of a graphical user interface, allowing the user to see the various component selections as well as information about each field. Using such an interface also gives an easy way to collect the additional information needed for the code generation, such as the plug-in name, output package, and output path.

The prototype created therefore provides an interactive graphical interface, instead of more standard command-line options. The main application window contains fields for entering all the plug-in creation parameters, as can be seen in figure 2. The window also displays which MIB files are currently loaded and a list of the field groups, called “tabs”. The MIB files and the tabs can then be further viewed in separate dialogs.



Plug-Out - Y:\OMA1\Tools\Plug-out\private\test\hcvg.plo

Plug-in name: HCVG

Base package: se.ericsson.etx.mtk.plugins.hcvg

Output path: Y:\OMA1\Tools\Plug-out\private\test\src

Tabs: Alarm, Channels, Control, Misc, OrigMod, Properties, Protocols, QosAddr, Statistics

MIBs: VOICEGATEWAYHC

Figure 2. The main application window contains global fields for the plug-in creation, as well as a list of all tabs and MIB files. The values of all fields can be saved and restored from the menu, allowing some level of iterations to the plug-in development.

Having a richer interaction also provides natural open and save functions, allowing plug-in generation parameters to be stored to file for later usage. This also provides some level of iteration to the plug-in development, as the previous choices can be stored until processing the next version of the MIB.

Structuring the fields

The structuring of the fields into tabs is difficult to do automatically, as the structure inside the MIB files does not provide a division that is recommendable to use for the interface. The tabs are therefore created manually in the prototype.

Tabs are constructed in two steps; first the tab is created and named, and second a set of fields is assigned to the tab. The tab creation is done in the lower left part of the main window (see figure 2), and new tabs can be created at any time. If a MIB file has been loaded, the fields from it can also be assigned to one of the tabs, something normally done in the tab edit dialog (see figure 3).

By default the fields don't belong to any tab, meaning that they will be ignored when generating the output plug-in. Each field may only belong to one single tab, and will thus be included in the output code for the plug-in and that specific tab.

Fields may also be loaded from several MIB files, which is sometimes necessary to allow common fields to be reused among several network components. An important restriction to the capability to load several MIB files in the current prototype is that no two fields may have the same name. If field names are equal, the generated output code may contain duplicate variable names.

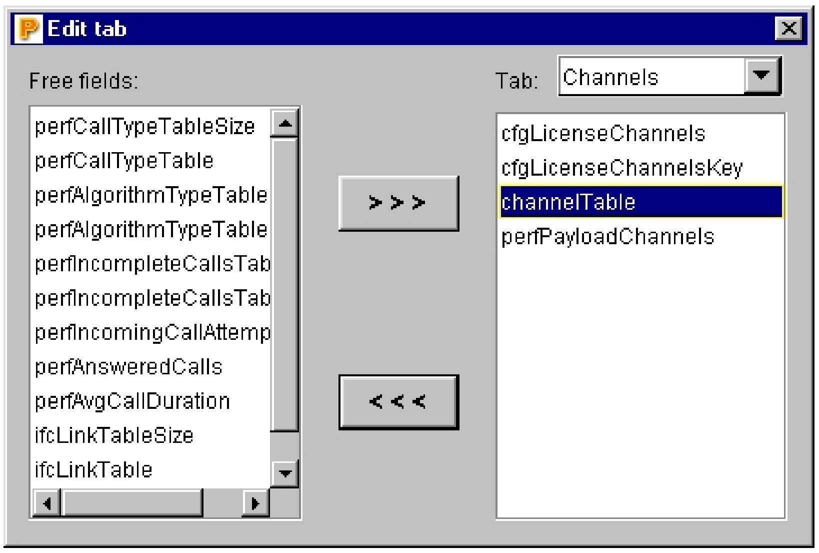


Figure 3. The tab edit dialog allows moving free fields to and from specific tabs. The list of free fields is ordered as the fields appear in the MIB files, and fields are removed when inserted into a tab.

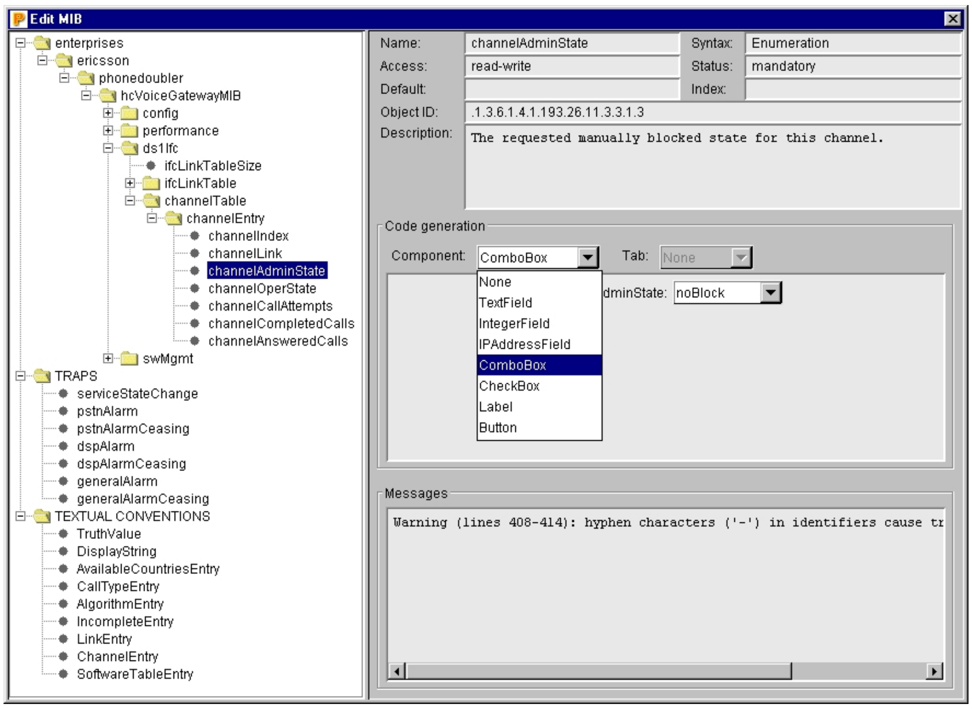
Changing the component mapping

Once a field has been assigned to a specific tab, a component is automatically chosen with the default field mapping presented previously. The chosen component can then be changed to any other component supported by the system, allowing the user to correct mistakes made in the automated process.

The changing of components is done in the MIB dialog (see figure 4), where all the fields in the MIB are presented along with analysis information. The tree structure of the MIB files is used for presenting the fields to the user, although this information is not used when actually generating the plug-in.

There is no one-to-one mapping between the components available for selection, and the components in the underlying Java Swing framework. The components available are really component generators, as they are also capable of generating source code for the component. The component generators also handle the creation of several components where needed, without the need for manual interaction. Several Swing components are also missing as they have little usage in a plug-in interface, and some new components have been added, such as an integer field and an IP address field. The user can also choose to not generate a component for a specific field.

When choosing to not generate a component, the field will be excluded from the graphical interface, but code will still be generated for lower layers in the plug-in. The exclusion of a field from the interface does not mean that the data will not be accessible, as it will still be present in the lower plug-in layers.



Name: channelAdminState
Syntax: Enumeration
Access: read-write
Status: mandatory
Default:
Index:
Object ID: .1.3.6.1.4.1.193.26.11.3.3.1.3
Description: The requested manually blocked state for this channel.

Component: ComboBox
Tab: None
dminState: noBlock

```
Warning (lines 408-414): hyphen characters ('-') in identifiers cause tr
```

Figure 4. The MIB edit dialog makes changing the selected component easy and straightforward. In addition most of the field information is displayed along with analysis messages.

4. Code Generation

This section explains how Java code for the plug-in is generated. The first part discusses general quality requirements on plug-in source code and how well these have been met in the prototype. The second part explains the design of the prototype code generator.

4.1 Generating source code of high quality

Generating source code is not trivial, as there are many dependencies between various files and code parts. The fact that the profession of programmers still exists (and is rather well paid) also indicates that automatically generated source code cannot solve all problems. In the case of generating plug-ins there are several such issues—user interface layout and logic, data interdependencies and program logic are all extremely hard to handle well in an automated fashion. The introduction of automated tools will not remove the need for plug-in developers, as several parts of the code must be inserted by hand.

In order to insert the missing parts, the generated Java source code must be possible to read, understand, and change by a plug-in programmer. Requiring that the programmer shall be able to read and understand the code is actually a rather hard requirement, as it means that the output code must be generally well-written and commented. The code must be of a high quality and should follow the same code conventions that the programmer is used to. In specifics this means that the generated code must be (in parts from Huss, 1993):

- General
- Modularized
- Indented
- Commented
- Correct
- Extensible
- Robust
- Compatible
- Efficient

The source code requirements

That the code be general means that it should be possible to use for as many purposes as possible. Not only do there exist several distinct types of plug-ins, but it may also be the case that other plug-in platforms may be used. The generated code must thus be as general as possible, making it possible to reuse for other purposes.

Modularization means dividing the generated code into methods and classes responsible for parts of the functionality. Together with the indentation and commenting this highly improves the readability of the code for the plug-in programmer.

Correctness means that the generated code must be possible to compile or that it at least is in such a state that it can be compiled with minimal additions. There should be no syntactic errors present in the code, and the semantics should of course also make sense, even if it the generated code is only a partial solution.

Extensibility means that the generated code shall be simple to extend with new functionality, either by changing or adding new methods. This requirement is only partially met by generating well-structured code, the other part needed is preserving these additions when generating the plug-in again.

Robustness means handling invalid user input in correct ways. The generated components must check their entry to the largest possible extent for errors, and should fail gracefully.

Compatibility means supporting the plug-in platform, something that in parts collide with the demand of keeping the generated code as general as possible. A special compatibility demand has been that the components should be possible to layout with the visual tools in Borland JBuilder 3. This means keeping the interface creation parts of the code within methods with particular names.

Finally the requirement of efficiency means that the generated code should not be any slower to execute than hand-made code. No additional lookups or runtime conversions can be tolerated in the output code—all such computations should be made already when generating the code.

Limits in the design

In an ideal world, the generated plug-in programs should always fulfill all of these requirements. However, in the real world some of these requirements cause conflicts or excessive work when generating the plug-ins. In the following the design decisions that have put a limit to these requirements are explained in more detail.

The generated plug-ins are not totally correct, as they lack several important features. Some of these features have been left out as they were outside the original scope of this thesis, i.e. not related to the GUI generation. Others have been left out as they proved impossible to handle by automated means, as the component layout. Many parts left out in this way have been commented in the output source code.

The code is not always possible to compile directly as a result of using some method names that are not declared in the code. The idea is to allow the programmer to choose either an adequate superclass or to implement the method manually. In order to make this choice more obvious, no default alternative is generated.

The generated code is not fully extensible, as it is hard to avoid overwriting the source code on disk. That kind of support requires parsing it and keeping track of user changes, something that would have been a far to large task to undertake.

As there was a conflict between making the code more general or fully compatible with the plug-in platform used today, the choice was made to not make the code dependent on the plug-in platform. This means that the generated classes does not inherit any specialized superclasses from the plug-in framework.

4.2 Layered code generation

There are basically two implementation strategies for automatic code generation. The output source code can either be partially written in separate files, called *templates*, or created totally by the program. Using templates usually makes it easier to maintain the code generator, as many changes can be made directly to the template files instead of in the generator.

Using templates when generating source code of high quality may cause problems, however, as it is important that the output be given a consistent formatting. It is also difficult to maintain the tight connection between the code generator and the templates, as these are split up into several files. The better solution is therefore to handle the code generation inside several specialized modules in the program.

An approach to generate the source code totally from within the program calls for some modularization, as it would otherwise be almost impossible to guarantee high quality of the output source code. The code generation in the prototype application has therefore been split into three layers—the Java syntax layer, the file or class layer, and the overall plug-in layer (see figure 5).

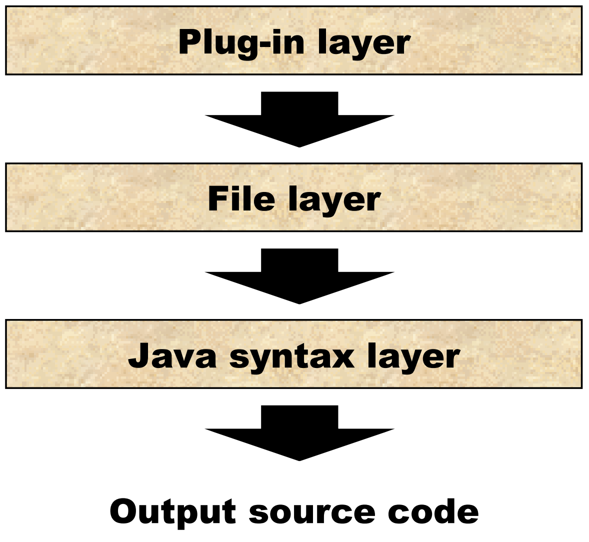


Figure 5. The source code generation is divided into three layers, each one only calling functions in the layer below. The syntax layer handles the syntactic constructs in Java, the file layer handles the dependencies between different files, and the plug-in layer controls what to generate.

The Java syntax layer

The Java syntax layer generates the actual text files containing the source code. It creates code for many of the basic syntactic constructs in the Java programming language—for example class, variable, and method declarations. The output text is also formatted following the recommendations in the Java coding style guide (Reddy, 1998).

The use of a separate syntax layer guarantees that all the generated code will follow the same well-defined code conventions, while at the same time making it easier to change style. As an example, the number of spaces for each indentation level can be changed by modifying only a single constant in the prototype. The interfaces provided also makes it easier to generate the source code, as much complexity and details can be hidden.

The syntax layer consists of a set of classes, each representing a syntactic construct in the language. From these classes the concrete instances can be created and combined in the same ways that the Java syntax allows, making it impossible to generate output code that is not syntactically valid for those constructs (see figure 6). The top object in such a hierarchy is always a Java source file, and when writing those objects to disk all substructures are also written.

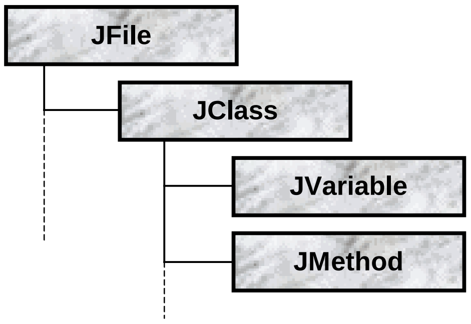


Figure 6. A simplified Java syntax structure. The file object can only have classes as children. The classes, in turn, can only accept variables and methods. These rules makes it harder to generate a non-valid Java file.

In order to keep the syntax layer more effective and compact, only top level declarations are handled as specialized objects. Declarations and the actual code inside methods are handled as simple strings, opening up for syntactic errors and misspellings. It would, however, be difficult to create all the code as a reversed parse tree, in particular as that would require creating lots of objects in other parts of the code.

The file layer

A plug-in may consist of a considerable amount of source code files, each containing one or more Java classes. In order to structure those files correctly, and keep track of changing variable and method names, a file layer is inserted on top of the syntax layer to handle this. The file layer also inserts code and comments that are the same for all plug-ins.

The file layer consists of a set of classes, each modeling a single output file. Unlike the syntax layer, the file objects may not be freely created and combined, and other layers can only explicitly create two files—the plug-in file and the tab file. These two automatically create all other files below them in the hierarchy, in order to assure that all dependencies become correct.

The file objects also allows some level of manipulation through methods that can add code to the various files. These calls are translated to the syntax layer, and each file object keeps an internal reference to the syntax layer objects needed. The file layer also communicates horizontally to a large extent, in order to assure that variable and method names are the same in the declaration and when using them.

By using a specialized file layer, the complicated dependencies between the files (as seen in figure 7) can be kept under control without much effort. The problems with exporting and importing variables and methods can also be approached in a safer manner, as each file object is responsible for creating the references to itself.

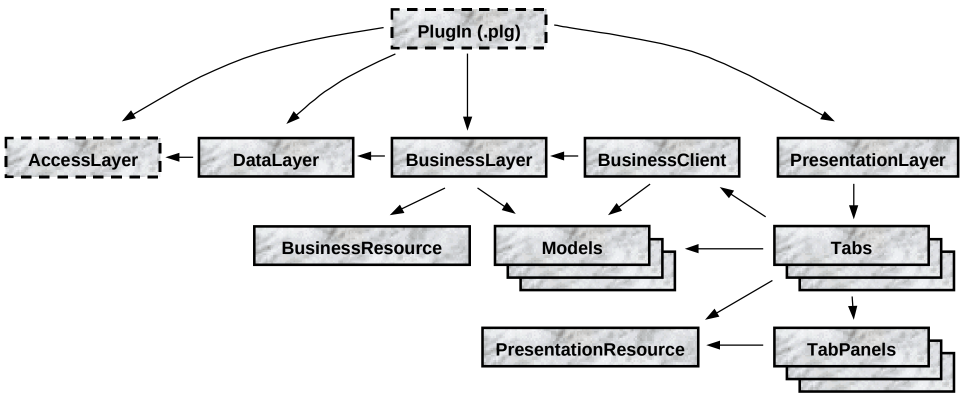


Figure 7. The structure of the plug-in source code files. Dependencies between files have been drawn as arrows from one class to another. Files with a dotted border are currently not generated by the prototype.

The plug-in layer

On top of the file layer the full plug-in generation is placed, handling such things as creating the tabs, the components, and everything that is related to the actual MIB fields. The major part of this layer consists of the component generation, as the prototype generates little more than skeleton code for the other parts.

The components and all related code are generated by component generator objects, making the component generation modularized and extensible. Apart from creating the actual source code, they also provide the simple demonstrations of the components that are available in the interface (seen in figure 4).

It is not trivial to add new components to the system, however, since each component is also responsible for generating its own source code. The number of component generators has therefore been limited in the prototype. Only the most commonly used components, such as text and number fields, have been implemented in the prototype (see table 2). Some extensions to the standard Java components have also been made in order to simplify the code generated.

The component generators can be freely interchanged, with the exception of the table component generator. They all implement a uniform interface and read the type information direct from the symbols, allowing each component generator to handle its own subset of the type semantics. Currently the prototype does not handle type conversions from the underlying data types to the types needed in the

Table 2. The component generators available in the prototype can only create a subset of the components in Java.

Component	Description
Label	A read-only text field
TextField	A text edit field
IntegerField	An integer edit field (not standard Java)
IPAddressField	A numeric IP address field (not standard Java)
CheckBox	A check box for true/false alternatives
ComboBox	A field that only allows one of a set of alternatives
Button	A push button
Table	A table of other components (not freely selectable)

components, since that conversion is complicated to generate automatically and is outside the scope of this thesis.

In the current prototype implementation several of the plug-in layers are not completely generated. The access layer and the plug-in specification files are not generated at all (see figure 7), partly because they have a trivial structure and are simple to write by hand. The data and business layers are only generated as stub code, leaving out important parts of data conversion and inner mechanics. Fixing this would probably require a substantial effort, especially if the goal is to keep the generated plug-ins as general and multi-purpose as possible.

5. Evaluation and future work

This section evaluates the results obtained and outlines the further work needed on the prototype. The first part mentions the known problems and simpler improvements remaining to be done. The second part discusses a number of future extensions that could make the tool more powerful. The last part concludes that it was possible to automate large parts of the plug-in creation.

5.1 Implementation improvements

There are several possible improvements to the current prototype implementation, both in the user interface and in some of the internal data structures. Some of these issues are known and have been detected in late stages of the development, leaving too little time to correct them. In the following, the known improvements are listed along with an estimation of how much time it would take to implement them. The time estimates refer to a developer familiar with the prototype code organization, algorithms and data structures.

Closing the dialogs (4 hours)

Neither the tab nor the MIB edit dialogs contain close buttons, something that may be confusing to some users. It would therefore be good if such buttons could be inserted without making the interface more complicated than it already is.

Reordering the tabs (1 day)

The tabs are currently sorted in alphabetical order, which is clearly not the order in which they should be generated to the plug-in. The ordering should instead be possible to change by dragging and dropping in the tab list.

Multiple MIB dialogs (1 day)

It should be possible to view various MIB files at a time, which means making the MIB dialog non-modal. It should not be possible to have several dialogs viewing the same MIB, which can happen in the current implementation if the user double-clicks the button. There are also some problems deciding what shall happen if the user unloads a MIB currently viewed or loads another plug-in. This must be solved before making the MIB dialog non-modal.

Better error handling (5 days)

Errors encountered in the MIB files are displayed immediately, but the original text is not shown. Storing the text inside the parse tree would require much additional work, but reading the file again is probably both possible and fast enough. Warnings should be handled in the same way, and should also be displayed immediately in a separate dialog or window. Ideally the application should launch a MIB editor allowing the user to correct the mistakes in the file.

Symbol data structure change (5 days)

The symbol data structure, used for handling the MIB fields and data types internally, is too closely modeled after the ASN.1 syntax. Some new models need to be defined for symbol and type information, making the symbols easier to use in the code generation. A third pass in the MIB analysis would have to be added, translating the old symbol model into the new one.

5.2 Future extensions

There are several parts remaining to be designed and implemented in the prototype application, especially to generate the data access and data logic layers. In the following, most of the possible extensions are listed along with an estimation of how much time it would take to implement them. The time estimates refer to a developer familiar with the prototype code organization, algorithms and data structures.

Modularize the plug-in data layer (3 days)

The data layer of the generated plug-in should not be a monolithic class, but split into separate modules. A better modularization would save much work and increase speed, as the data is currently transferred in whole blocks to the network element currently configured. This separation could be made from the branches in the MIB, but might require that the internals of the MIB symbols be somewhat improved.

Convert field data to component data (unknown)

The low-level data representation is not always the same as the one used in the components. Some kind of intelligent type conversion must therefore be inserted in the business layer of the generated plug-in. Of course such a conversion would not be able to handle all conversions correctly, as it is not always clear what the mapping would be.

Generating user interface tooltips (5 days)

The generated plug-in user interface should contain tooltips, i.e. explanatory texts that are displayed when the user hoovers over the component with the pointer. This information could be extracted automatically from the field comments in the MIB, but must be possible to edit as the comments are not always suitable. This of course also means changing the file format when storing plug-in generation data, as the new text would have to be stored along with other information.

Editing the interface labels (3 days)

The labels that some components have in the interface should be editable. This of course also means changing the file format when storing plug-in generation data, as the new text would have to be stored along with other information.

One field in various tabs (4 days)

A field from the MIB files should be possible to include in various tabs, as it is needed in some plug-ins. Apart from changing inside the symbols, this would also require that the tab edit dialog be reimplemented to support an additional view of the fields. In the code generation special care would also have to be taken not to duplicate information in the lower plug-in layers.

Preview of tabs (2 days)

Currently example components are shown for each field, but it would also be good to be able to test the whole plug-in interface and all the tabs simultaneously. Ideally such a preview would also allow dragging the components, thereby generating a draft layout.

Generating SNMP agent (unknown)

The SNMP requests sent by the plug-in client are received by an agent (or server) in the other end of the communication. This agent is normally written in C and contains separate functions for each of the fields in the MIB. It would be of great value if stub code for these functions could be generated automatically, assuring that all fields used by the plug-in would also be handled by the SNMP agent. The most time consuming part of this would probably be creating a C syntax layer in the code generation.

5.3 Evaluation of the results

The overall results from the prototype have been very encouraging—a user interface could be created from the MIB files and the lower layers of the plug-in seem viable for automation. The prototype application addresses several issues not originally planned, as the creation of language resource files and data layer stubs, and produces code of a higher quality than originally thought possible.

Most of the troubles have been caused by the MIB parsing, something not anticipated before embarking on it. The component selection, on the other hand, proved almost trivial once the right approach was taken. The code generation layering was developed in an iterative fashion and was very successful in solving many of the problems with dependencies in the code.

Somewhat discouraging, though, has been the weak support from the organization. The plug-in developers have shown little interest in the prototype, resulting in a product that has not been tested in a real-world setting. Several bugs and mistakes will probably be revealed once such testing is actually performed.

6. References

Berk, Elliot (1995); *JLex: A lexical analyzer generator for Java*, Users manual for version 1.2, <`http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html`> visited November 1999

Bosch, Jan (1996); “Delegating Compiler Objects – an Object-Oriented Approach to Crafting Compilers”, *Compiler construction: 6th International Conference*, Tibor Gyimóthy (editor), Springer Verlag, Berlin

Case, Fedor, Schoffstall & Davin (1990); *A Simple Network Management Protocol (SNMP)*, RFC 1157, <`http://www.ietf.org/rfc/rfc1157.txt`> visited September 1999

Gagnon, Éntienne (1998); *SableCC – an object-oriented compiler framework*, School of Computer Science, McGill University, Montreal, <`http://www.sable.mcgill.ca/sablecc/`> visited September 1999

Huss, Håkan (1993); *Grundläggande programkvalitet* (Fundamentals of Program Quality), from the course material to “Introduktion till Datalogi (2D1340)” (Introduction to Computer Science) at the Department of Numerical Analysis and Computing Science (Nada), Royal Institute of Technology (KTH)

ISO 8824; *Information technology – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*, International Organization for Standardization, International Standard 8824, 1990

Reddy, Achut (1998); *A Coding Style Guide for Java WorkShop and Java Studio Programming*, Sun Microsystems, <`http://www.sun.com/workshop/java/wp-coding/`> visited December 1999

Rinderknecht, Christian (1995); *Parsing ASN.1:1990 with Caml Light*, INRIA Technical Report #171, <`http://cristal.inria.fr/~rinderknecht/publis.html`> visited September 1999

Rose, M. (1991); *A Convention for Defining Traps for use with the SNMP*, RFC 1215, <`http://www.ietf.org/rfc/rfc1215.txt`> visited September 1999

Rose & McCloghrie (1990); *Structure and Identification of Management Information for TCP/IP-based Internets*, RFC 1155, <`http://www.ietf.org/rfc/rfc1155.txt`> visited September 1999

Rose & McCloghrie (1991); *Concise MIB Definitions*, RFC 1212, <`http://www.ietf.org/rfc/rfc1212.txt`> visited September 1999

Sun Microsystems, Inc (1999); *JavaCC 1.0 Documentation*, <`http://www.metamata.com/JavaCC/`> visited September 1999

A. Appendices

A.1 Simplified ASN.1 grammar

```
/*
 * Simplified EBNF grammar for ASN.1:90
 *
 * Version:   1.2
 * Date:      10th of August 1999
 * Author:    Per Cederberg, qtxpece@etx.ericsson.se
 *
 * This grammar has been extracted from the yacc and lex
 * sources of ’snacc’, a GNU ASN.1 to C or C++ compiler by
 * Mike Sample.  It has then been modified and cleaned,
 * making it simpler and more compact. Some additional
 * types commonly used in Internet MIBs have also been
 * added.
 */


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  Module def/import/export productions
 *
 */

Start ::= ModuleDefinition

ModuleDefinition ::= ModuleIdentifier "DEFINITIONS" [TagDefault]
                     "::=" "BEGIN" [ModuleBody] "END"

TagDefault ::= "EXPLICIT" "TAGS"
             | "IMPLICIT" "TAGS"

ModuleIdentifier ::= ModuleReference [ObjectIdentifierValue]

ModuleBody ::= [Exports] [Imports] AssignmentList

Exports ::= "EXPORTS" [SymbolList] ";"

Imports ::= "IMPORTS" [SymbolsFromModuleList] ";"

SymbolsFromModuleList ::= (SymbolsFromModule)+

SymbolsFromModule ::= SymbolList "FROM" ModuleIdentifier

SymbolList ::= Symbol ("," Symbol)*

Symbol ::= TypeReference
         | Identifier
         | DefinedMacroName
```

```
AssignmentList ::= (Assignment [";"])+

Assignment ::= TypeAssignment
             | ValueAssignment
             | MacroDefinition

MacroDefinition ::= MacroReference "MACRO" "::=" MacroBody

MacroBody ::= "BEGIN" <SkipToEND> "END"
            | ModuleReference "." MacroReference

MacroReference ::= TypeReference
                 | DefinedMacroName



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  Type Notation Productions
 *
 */

TypeAssignment ::= TypeReference "::=" Type

Type ::= BuiltinType
       | DefinedType
       | DefinedMacroType

BuiltinType ::= "BOOLEAN"
              | "REAL"
              | "NULL"
              | "OBJECT" "IDENTIFIER"
              | IntegerType
              | StringType
              | BitStringType
              | SequenceType
              | SequenceOfType
              | SetType
              | SetOfType
              | ChoiceType
              | EnumeratedType
              | SelectionType
              | TaggedType
              | AnyType

IntegerType ::= "INTEGER" [NamedNumberList | ConstraintList]

NamedNumberList ::= "{" NamedNumber ("," NamedNumber)* "}"

NamedNumber ::= Identifier "(" (SignedNumber|DefinedValue) ")"

SignedNumber ::= ["-"] Number

StringType ::= "OCTET" "STRING" [ConstraintList]

BitStringType ::= "BIT" "STRING"
                  [NamedNumberList | ConstraintList]
```

```
SequenceType ::= "SEQUENCE" "{" [ElementTypeList] "}"

SequenceOfType ::= "SEQUENCE" [SizeConstraint] "OF" Type

SetType ::= "SET" "{" [ElementTypeList] "}"

SetOfType ::= "SET" [SizeConstraint] "OF" Type

ElementTypeList ::= ElementType ("," ElementType)*

ElementType ::= NamedType
              | NamedType "OPTIONAL"
              | NamedType "DEFAULT" NamedValue
              | [Identifier] "COMPONENTS" "OF" Type

NamedType ::= [Identifier] Type

ChoiceType ::= "CHOICE" "{" ElementTypeList "}"

EnumeratedType ::= "ENUMERATED" NamedNumberList

SelectionType ::= Identifier "<" Type

TaggedType ::= Tag ["IMPLICIT"|"EXPLICIT"] Type

Tag ::= "[" [Class] ClassNumber "]"

ClassNumber ::= Number
              | DefinedValue

Class ::= "UNIVERSAL"
        | "APPLICATION"
        | "PRIVATE"

AnyType ::= "ANY" ["DEFINED" "BY" Identifier]

DefinedType ::= [ModuleReference "."] TypeReference
                [ConstraintList]

ConstraintList ::= "(" Constraint ("|" Constraint)* ")"

Constraint ::= ValueConstraint
             | SizeConstraint
             | AlphabetConstraint

ValueConstraint ::= Value
                  | ValueRange

ValueRange ::= LowerEndPoint ".." UpperEndPoint

LowerEndPoint ::= (Value|"MIN") ["<"]

UpperEndPoint ::= ["<"] (Value|"MAX")

SizeConstraint ::= "SIZE" "(" ValueConstraint ")"

AlphabetConstraint ::= "FROM" "(" ValueConstraint ")"
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  Value Notation Productions
 *
 */

ValueAssignment ::= Identifier Type "::=" Value

Value ::= BuiltinValue
        | DefinedValue

DefinedValue ::= [ModuleReference "."] Identifier

BuiltinValue ::= BooleanValue
               | NullValue
               | SpecialRealValue
               | SignedNumber
               | HexString
               | BinaryString
               | CharString
               | ObjectIdentifierValue

BooleanValue ::= "TRUE"
               | "FALSE"

NullValue ::= "NULL"

SpecialRealValue ::= "PLUS-INFINITY"
                   | "MINUS-INFINITY"

NamedValue ::= [Identifier] Value

ObjectIdentifierValue ::= "{" ObjIdComponentList "}"

ObjIdComponentList ::= (ObjIdComponent)+

ObjIdComponent ::= Number
                 | Identifier
                 | NameAndNumberForm

NameAndNumberForm := Identifier "(" Number ")"
                   | Identifier "(" DefinedValue ")"

BinaryString ::= "’" <BinaryChars>* "’B"

HexString ::= "’" <HexadecimalChars>* "’H"

CharString ::= """ <StringWithoutSingleDoubleQuote> """

Number ::= <PositiveNumber>

Identifier ::= <LowerCaseFirstIndentifier>

ModuleReference ::= <UpperCaseFirstIdentifier>

TypeReference ::= <UpperCaseFirstIdentifier>
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  Macro Syntax definitions
 *
 */

DefinedMacroType ::= SnmpObjectTypeMacroType
                   | SnmpTrapTypeMacroType

DefinedMacroName ::= "OBJECT-TYPE"
                   | "TRAP-TYPE"

SnmpObjectTypeMacroType ::= "OBJECT-TYPE"
                            "SYNTAX" Type
                            SnmpAccessPart
                            SnmpStatusPart
                            [SnmpDescrPart]
                            [SnmpReferPart]
                            [SnmpIndexPart]
                            [SnmpDefValPart]

SnmpTrapTypeMacroType ::= "TRAP-TYPE"
                          "ENTERPRISE" Identifier
                          [SnmpVarPart]
                          [SnmpDescrPart]
                          [SnmpReferPart]

SnmpAccessPart ::= "ACCESS" Identifier

SnmpStatusPart ::= "STATUS" Identifier

SnmpDescrPart ::= "DESCRIPTION" CharString

SnmpReferPart ::= "REFERENCE" CharString

SnmpIndexPart ::= "INDEX" "{" TypeOrValueList "}"

TypeOrValueList ::= TypeOrValue ("," TypeOrValue)*

TypeOrValue ::= Type
              | Value

SnmpDefValPart ::= "DEFVAL" "{" Value "}"

SnmpVarPart ::= "VARIABLES" "{" VarTypes "}"

VarTypes ::= Identifier ("," Identifier)*
```